

Selenium and Appium with Python

**Build robust and scalable test automation frameworks
using Selenium, Appium and Python**



Yogashiva Mathivanan





Selenium and Appium with Python

*Build robust and scalable test automation
frameworks using Selenium, Appium and Python*

Yogashiva Mathivanan



www.bpbonline.com

Copyright © 2023 BPB Online

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor BPB Online or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

BPB Online has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, BPB Online cannot guarantee the accuracy of this information.

First published: 2023

Published by BPB Online

WeWork

119 Marylebone Road

London NW1 5PU

UK | UAE | INDIA | SINGAPORE

ISBN 978-93-55518-354

Dedicated to

My beloved Parents:

Shri Mathivanan Gurusamy

Sita Kani Mathivanan

&

*My brother Guru Prakash Mathivanan,
My wife Hymanivedita Yogashiva, and
My son Adyant Prakash Yogashiva*

About the Author

Yogashiva Mathivanan is a highly accomplished Test Automation Architect with a Master's degree in Computer Engineering from New York, USA. He has over 10 years of experience in Software Test Automation and Framework design. Yogashiva has delivered top-quality test automation solutions to companies such as Infosys, IBM, HCL, MoneyGram, LPL Financial, T-Mobile, Experian, and Coinbase, and is currently working at CVS Health. Yogashiva has demonstrated exceptional proficiency in Java and Python programming languages and has an excellent track record of designing and implementing robust automation frameworks to automate Web, API, and Mobile Applications. Mathivanan's expertise also extends to data validation across diverse databases & data applications, performance, and security testing. Mathivanan's exceptional skills in problem-solving, attention to detail, and team management helped the companies deliver projects with the highest quality standards and have earned him a reputation as a highly reliable and effective leader in the software testing industry.

About the Reviewer

Animesh is a Test Automation Specialist with over 9.5 years of experience in India, Australia and Europe, across multiple technologies and domains. He loves talking about Python, Selenium, Appium, AWS, APIs, Docker, Kubernetes, GIT, SQL and many other technologies. He is currently working in Poland for a renowned corporate.

Acknowledgements

I feel grateful and want to thank my father, Mathivanan, my mother, Sita Kani, my brother, Guru Prakash, my wife, Hymanivedita, and my son, Adyant Prakash, for their support and inspiration throughout the writing of this book. Their encouragement and belief in my abilities helped me navigate through the challenges I encountered along the way. Their presence in my life has been a blessing.

I am also deeply grateful to Priyadarshini Institute of Technology, Nagpur, India where I completed my bachelor's degree, and New York Institute of Technology, NY, USA, where I earned my master's degree, for providing me with the knowledge and skills, that laid the foundation for this book. Moreover, I am thankful to the companies and co-workers I worked with in the software industry, who have given me invaluable experience and the confidence to pursue this project.

Finally, I would like to express my gratitude to the BPB Publications team for their support and consideration during the challenging moments I encountered while writing this book. Additionally, their guidance and expertise were instrumental in bringing out the best version of this book. Furthermore, I extend my heartfelt thanks to all the readers who have taken an interest in my work. I hope this book provides you with the tools and knowledge to help you advance in your career and achieve your goals.

Preface

Automation testing has become an essential component of software testing. The ability to automate tests not only improves efficiency but also reduces the risk of human error. This book provides a comprehensive guide to automation testing using Python programming language and popular automation testing tools Selenium and Appium, for Web and Mobile applications, respectively.

This book covers the fundamentals of automation testing and its role in testing. It then introduces Python programming for automation testing and explores Selenium and Appium for web and mobile app automation, including handling web elements, locators, and gestures on mobile. Advanced topics such as synchronization, exception handling, assertions, and hybrid application in mobile are also discussed.

The book also includes coverage of designing automation frameworks for web and mobile applications from scratch, Docker & Selenium grid, and a bonus chapter on Python interview questions to help readers in interview preparation.

This book is suitable for both beginners and experienced automation testers. It offers a practical approach and real-world examples to help you gain a deep understanding of automation testing and frameworks, enabling you to advance your automation testing career.

The book is structured into thirteen chapters that cover all aspects of automation testing, from the basics of testing and automation process, and framework design, to advanced concepts such as Dockerized Selenium Grid. The details are listed below.

Chapter 1: Testing Process and Role of Automation – provides key software testing and process concepts essential for interviews. The chapter sets the foundation for the software testing process and automation, and explains the significance of software testing in ensuring the quality & reliability of the application, and other benefits. The chapter covers the core concepts such as the different types of software testing and terms in software testing, the evolution of software testing with the Software Development Life Cycle (SDLC) models and modern Agile Methodology, defect/bug life cycle & different states of a defect, and the role and significance of automation in software testing.

Chapter 2: Python Programming - Setup and Core Concepts – provides fundamental concepts of the Python programming language, starting from installation/setup to core concepts such as variables, data types, expressions, control flow statements, and loops. The chapter also covers important data structures like lists, tuples, sets, and dictionaries, as well as functions in Python. This chapter sets the foundation for Python programming language and is essential for proceeding with software automation using Selenium and Appium with Python in the following chapter.

Chapter 3: Selenium for Web Automation – introduces the Selenium tool, covering its architecture, installation, and setup. It explains how to invoke browsers using Selenium Webdriver with and without webdriver manager, and configure the invoked browser's window, followed by the automation of the first scenario. The chapter details the key differences between Selenium 3 and 4 versions, along with the benefits of using Selenium for web automation with Python, emphasizing the popularity of Python + Selenium.

Chapter 4: Appium for Mobile Automation – introduces the Appium tool, its architecture, advantages, installation, and setup of appium and supporting tools like Appium Inspector, Android Studio, XCode, and so on. The chapter explains how to install and launch an Android application, configure an iOS simulator to launch an IOS application, and invoke and launch a mobile application and locate elements using Appium Inspector. The chapter details Appium's popularity and versatility in automating native, mobile web, and hybrid applications.

Chapter 5: Locators and Handling Web Elements – explains how to find and interact with both basic and advanced web application elements using Selenium Locators. The chapter provides a detailed explanation of these locators and how to construct and validate them, as well as information on useful browser extensions and finding multiple web elements. Additionally, readers will explore advanced web elements scenarios such as working with web tables, iFrames, window handles, and advanced operations like drag & drop and double click. The chapter emphasizes the correct usage of locators and understanding the HTML and DOM structure of the web page, to build stable automation scripts.

Chapter 6: Appium: Locators and Gestures – explains how to locate and interact with mobile application elements using Appium’s locators and gestures. It covers various element locator strategies such as UIAutomator, AccessibilityID, ID, Class Name, Name, and Xpath, along with how to use the Android Inspector to locate elements and their properties. The chapter also covers essential mobile-specific driver methods, Android keycode usage, and mobile element properties/attributes. Furthermore, readers will learn how to perform different gestures such as tap, long press, swipe, and scroll gestures and how to automate them using Appium.

Chapter 7: Synchronization, Exception Handling and Assertions – provides an in-depth understanding of synchronization methods in Selenium, including unconditional and conditional synchronization, such as implicit, explicit, and fluent waits. It also covers common exceptions in Selenium and how to handle them with a try-catch block. Additionally, the chapter introduces assertions, which are utilized to validate the behavior of elements. Readers will learn how to ensure reliable and stable test execution in Selenium WebDriver.

Chapter 8: Hybrid Application Automation & Launching Multiple Apps – covers the automation of hybrid applications using Appium for both Android and iOS devices. It explains how hybrid apps consist of both native and web components, and how to identify and switch between the different contexts, such as native app and WebView, to perform actions on elements. Additionally, the chapter covers how to switch between multiple apps during execution. Overall, readers can expect to gain knowledge on how to automate hybrid apps.

Chapter 9: Selenium Automation Framework – Part 1 – introduces readers to the automation framework and the importance of choosing the right framework for a testing project. It covers various types of frameworks, their design, organizing scripts, and the use of Python packages to enhance their stability. The chapter discusses in detail the four popular Python testing frameworks, namely PyTest, Robot Framework, Unittest, and Behave, with particular emphasis on PyTest and explaining their unique features, strengths, and use cases.

Chapter 10: Selenium Automation Framework – Part 2 – focuses on implementing the Page Object Model (POM) for UI automation, covering important topics such as taking screenshots, PyAutoGUI module usage, working with configurations to manage test data, logging, parallel test execution using pytest-xdist, os & pathlib for file system management, and data-driven testing using Python packages such as NumPy, Pandas, CSV, and openpyxl. A well-structured folder hierarchy is also emphasized to help organize test files, resources, and configurations. Additionally, the chapter highlights the essential Python modules and libraries used in test automation, such as datetime, random, and faker.

Chapter 11: Mobile Automation Framework – discusses the design and implementation of a mobile automation framework using Appium. It begins with introducing the Allure reporting tool, which is used to generate detailed reports with additional features, and then it covers the various components of the Appium framework, including folder structure, driver initialization, base page, configuration, utilities, pages, and tests.

Chapter 12: Dockerized Selenium Grid – introduces Docker as a containerization platform and its advantages over traditional virtualization. The chapter covers essential Docker terminologies, and commands, and how to install and set up Docker to run Selenium tests and Selenium Grid in Docker containers. Readers will learn how to use Docker Compose to define and run multi-container Docker applications.

Chapter 13: Bonus Chapter - Python Interview Questions – contains a collection of basic and intermediate-level Python programming interview questions designed for beginners and junior engineers, as well as senior engineers. Readers can gain confidence in their Python programming skills and increase their chances of landing a job in a test automation role.

Code Bundle and Coloured Images

Please follow the link to download the *Code Bundle* and the *Coloured Images* of the book:

<https://rebrand.ly/g4bost0>

The code bundle for the book is also hosted on GitHub at <https://github.com/bpbpublications/Selenium-and-Appium-with-Python>. In case there's an update to the code, it will be updated on the existing GitHub repository.

We have code bundles from our rich catalogue of books and videos available at <https://github.com/bpbpublications>. Check them out!

Errata

We take immense pride in our work at BPB Publications and follow best practices to ensure the accuracy of our content to provide with an indulging reading experience to our subscribers. Our readers are our mirrors, and we use their inputs to reflect and improve upon human errors, if any, that may have occurred during the publishing processes involved. To let us maintain the quality and help us reach out to any readers who might be having difficulties due to any unforeseen errors, please write to us at :

errata@bpbonline.com

Your support, suggestions and feedbacks are highly appreciated by the BPB Publications' Family.

Did you know that BPB offers eBook versions of every book published, with PDF and ePUB files available? You can upgrade to the eBook version at www.bpbonline.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at :

business@bpbonline.com for more details.

At www.bpponline.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on BPB books and eBooks.

Piracy

If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at business@bpbonline.com with a link to the material.

If you are interested in becoming an author

If there is a topic that you have expertise in, and you are interested in either writing or contributing to a book, please visit www.bpbonline.com. We have worked with thousands of developers and tech professionals, just like you, to help them share their insights with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions. We at BPB can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about BPB, please visit www.bpbonline.com.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

[https://discord\(bpbonline\).com](https://discord(bpbonline).com)



Table of Contents

1. Testing Process and Role of Automation.....	1
Introduction.....	1
Structure.....	2
Objectives.....	2
Significance of software testing	2
Core concepts of software testing	4
<i>Software Testing Process.....</i>	6
Types of software testing.....	6
<i>Manual testing.....</i>	7
<i>Advantages of manual testing.....</i>	7
<i>Disadvantages of manual testing.....</i>	8
<i>Functional and non-functional testing.....</i>	8
<i>Unit testing</i>	8
<i>White-Box, Black-Box, and Grey-Box Testing.....</i>	9
<i>Integration testing</i>	9
<i>User acceptance testing</i>	9
<i>Alpha and beta testing</i>	9
<i>System testing</i>	9
<i>End-to-end testing.....</i>	9
<i>Sanity testing</i>	9
<i>Smoke testing.....</i>	10
Software Development Life Cycle (SDLC).....	10
<i>Waterfall model.....</i>	10
<i>Waterfall model application</i>	11
<i>Advantages of the Waterfall model.....</i>	11
<i>Disadvantages of the waterfall model</i>	11
<i>Iterative (and incremental) model.....</i>	12
<i>Iterative and incremental model application</i>	12
<i>Advantages of the Iterative and Incremental Model.....</i>	13
<i>Disadvantages of the iterative and incremental model.....</i>	13
<i>Agile methodology – SCRUM</i>	13

<i>SCRUM Agile methodology application</i>	14
<i>Advantages of SCRUM Agile methodology</i>	14
<i>Disadvantages of SCRUM Agile Methodology</i>	14
Defect/Bug Life Cycle	15
<i>Different states of defect</i>	15
Automation in software testing	16
<i>Advantages of automation testing</i>	17
Conclusion	17
Key facts.....	18
Questions.....	18
2. Python Programming - Setup and Core Concepts.....	19
Introduction.....	19
Structure	20
Objectives.....	20
Advantages of Python.....	21
Getting started with Python	21
<i>Installation of Python in Windows</i>	21
<i>Installation of Python in Mac OS</i>	25
<i>Installation of PyCharm</i>	27
The Python interpreter	28
Executing the first Python code	28
<i>Using an Interactive Interpreter</i>	28
<i>Using a command line</i>	29
<i>Using an IDE: PyCharm</i>	30
<i>Understanding __name__ = “__main__”</i>	31
Virtual environment and Requirement.txt file.....	31
<i>Requirement.txt</i>	33
Core concepts of Python	34
<i>Variables and Data Types</i>	34
<i>Types of operators in Python</i>	36
<i>Arithmetic operator</i>	36
<i>Comparison operator</i>	37
<i>Assignment operator</i>	37

<i>Logical, identity, and membership operator</i>	38
<i>Lists in Python</i>	38
<i>Tuples</i>	40
<i>Sets</i>	40
<i>Dictionaries</i>	42
<i>Conditional statements in Python</i>	43
<i>IF-ELSE Condition</i>	43
<i>Indentation</i>	43
<i>Loops in Python</i>	44
<i>WHILE Loop</i>	44
<i>FOR Loop</i>	46
<i>FOR ELSE loop</i>	46
<i>Functions in Python</i>	47
<i>Global variables and local variables</i>	47
<i>Modules, packages, exception handling in Python</i>	49
<i>Modules</i>	49
<i>Packages</i>	50
<i>Errors and exceptions in Python</i>	51
<i>Syntax Error</i>	51
<i>Exceptions</i>	51
<i>Exception Handling</i>	52
<i>Classes and Objects in Python</i>	54
<i>Inheritance in Python</i>	55
<i>Conclusion</i>	57
<i>Key facts</i>	57
<i>Questions</i>	57
3. Selenium for Web Automation	59
<i>Introduction</i>	59
<i>Structure</i>	60
<i>Objectives</i>	60
<i>What is Selenium?</i>	60
<i>Advantages of Selenium</i>	62
<i>Selenium compared to other testing tools</i>	63

Selenium Webdriver Architecture	63
<i>Layers of Web application</i>	64
Selenium 3 architecture.....	64
Selenium 4 architecture.....	66
<i>Selenium 4 advantages</i>	67
Selenium WebDriver installation and setup.....	67
Basic Code: Browser Invoke and Windows setup.....	72
<i>Browser window setup</i>	74
First automation: Login scenario	75
<i>Deprecated Selenium 3 code</i>	75
<i>Selenium 4 test script</i>	76
<i>Difference between driver.close() and driver.quit()</i>	77
Webdriver Manager	77
<i>Before webdriver manager in Selenium 3</i>	77
<i>Before webdriver manager in Selenium 4</i>	77
<i>After webdriver manager in Selenium 3</i>	78
<i>After webdriver manager in Selenium 4</i>	78
Webdriver manager versus driver path	78
Conclusion	79
Key facts.....	79
Questions.....	80
4. Appium for Mobile Automation.....	81
Introduction.....	81
Structure	82
Objectives	82
Appium tool for mobile automation.....	82
<i>Mobile application testing</i>	82
<i>Types of Mobile Testing</i>	83
<i>Appium architecture</i>	83
<i>Appium drivers</i>	85
<i>Advantages of Appium</i>	85
Appium setup on Windows	86
Appium setup on Mac	92

Android Emulator configuration on Mac and Windows.....	96
<i>Installation of APK File</i>	98
<i>Appium Desired Capabilities</i>	99
Application Launching: Android	103
<i>Appium Server Launch Programmatically</i>	105
<i>Appium Server launch command prompt/terminal</i>	106
IOS Simulator Configuration on Mac	107
<i>Application launching: iOS</i>	108
Conclusion	109
Key facts.....	110
Questions.....	110
5. Locators and Handling Web Elements	111
Introduction.....	111
Structure	111
Objectives.....	112
Locators in Selenium	112
<i>Web Elements in DOM</i>	113
Types of locators in Selenium	114
<i>Locating Element using ID</i>	115
<i>Locating Element using Class Name</i>	116
<i>Locating Element using Name Attribute</i>	117
<i>Locating Element using Link Text</i>	118
<i>Locating Element using Partial Link Text</i>	118
<i>Locating Element using Tag Name</i>	118
<i>Locating Element using CSS Selector</i>	119
<i>ID</i>	119
<i>CLASS</i>	120
<i>ATTRIBUTE and MULTIPLE ATTRIBUTES</i>	121
<i>Relative CSS Selector</i>	122
<i>Locating Element using XPath Locator</i>	122
<i>Absolute XPath</i>	123
<i>Relative XPath</i>	123
<i>XPath with logical operators and XPath Functions</i>	124
<i>XPath Axes</i>	126

<i>Validating XPath and CSS in Browser Console</i>	128
Chrome Locators Extensions.....	129
Finding multiple web elements and commands	130
<i>Commands</i>	130
Advanced web page elements	132
<i>Checkbox</i>	132
<i>Radio button</i>	132
<i>Dropdown</i>	133
<i>IFrame</i>	134
<i>Alert popup</i>	135
<i>Window Handles</i>	137
<i>Web tables</i>	138
<i>Action Chains for Advanced Operations</i>	139
Conclusion	140
Key facts.....	141
Questions.....	142
6. Appium: Locators and Gestures	143
Introduction.....	143
Structure	144
Objectives	144
Element locator strategy.....	144
<i>Finding element by UIAutomator</i>	145
<i>Finding Element by AccessibilityID</i>	146
<i>Finding Element by ID</i>	148
<i>Finding Element by Class Name</i>	148
<i>Finding Element by name</i>	149
<i>Finding Element by Xpath</i>	149
Find Elements Method.....	150
Miscellaneous Driver Methods	151
Android Keycodes.....	151
Element Properties/Attributes.....	152
Automating Gestures	155
<i>Tap Gesture</i>	155

<i>Long press gesture</i>	157
<i>Swipe and scroll gesture</i>	157
Conclusion	160
Key facts.....	160
Questions.....	161
7. Synchronization, Exception Handling and Assertions	163
Introduction.....	163
Structure.....	164
Objectives.....	164
Synchronization.....	165
<i>Unconditional Synchronization</i>	165
<i>Conditional synchronization</i>	165
<i>Implicit wait</i>	165
<i>Explicit wait</i>	167
<i>Fluent wait</i>	167
Selenium exceptions and handling	170
<i>Exceptions</i>	170
<i>Exception Handling</i>	172
Assertions.....	174
<i>Python Assert Statement</i>	175
Conclusion	178
Key facts.....	178
Questions.....	179
8. Hybrid Application Automation & Launching Multiple Apps	181
Introduction.....	181
Structure.....	182
Objectives.....	182
Hybrid application	182
<i>Hybrid mobile app examples</i>	183
<i>WebView</i>	183
<i>Android Hybrid App Automation</i>	186
<i>Identifying WebView elements</i>	186

<i>Switching to WebView Context to perform the required action and switching back to Native App</i>	189
<i>Troubleshooting of possible error related to chrome version.....</i>	191
<i>IOS Hybrid App Automation.....</i>	191
Switching between multiple applications.....	193
Conclusion	196
Key facts.....	196
Questions.....	197
9. Selenium Automation Framework – Part 1.....	199
Introduction.....	199
Structure	200
Objectives.....	201
Automation framework.....	201
<i>Why Automation framework?.....</i>	201
<i>Benefits of Automation framework</i>	202
<i>Types of automation framework.....</i>	204
<i>Linear scripting framework</i>	204
<i>Modular testing framework</i>	205
<i>Data-Driven testing framework.....</i>	205
<i>Keyword-driven testing framework</i>	206
<i>Behavior-driven development framework.....</i>	206
<i>Hybrid testing framework.....</i>	207
<i>Choosing the right automation framework.....</i>	207
Pytest.....	208
<i>Install Pytest</i>	208
<i>Writing test with Pytest.....</i>	209
<i>Executing Pytest from PyCharm and Command Line</i>	210
<i>Markers in Pytest</i>	213
<i>Fixtures in Pytest.....</i>	215
<i>Reusing Fixtures in Pytest.....</i>	216
<i>Parameterization in fixtures in Pytest.....</i>	219
<i>Soft Assert in pytest – Softest</i>	220
<i>HTML reports in Pytest using pytest-html.....</i>	222
<i>pytest-html Install</i>	223

<i>Generate pytest-html report.....</i>	223
<i>View the report.....</i>	223
<i>Advantages of Pytest</i>	224
Robot framework	224
<i>Why Robot framework?</i>	224
<i>High-level architecture</i>	225
<i>Robot framework installation.....</i>	226
<i>Robot Framework Libraries</i>	227
<i>Robot Framework keywords</i>	228
<i>Robot Framework Folder Structure.....</i>	229
<i>Folder structure best practice.....</i>	230
<i>Test Data Management in Robot Framework.....</i>	231
<i>Robot framework test execution flow</i>	233
<i>Test setup/Test suite setup.....</i>	233
<i>Webdrivermanager</i>	234
<i>Test case execution</i>	234
<i>Test teardown/Test suite teardown.....</i>	235
<i>Logs and reports in Robot framework.....</i>	238
<i>Advantages of Robot framework.....</i>	239
Unittest	240
Behave.....	243
<i>Behavior Driven Development Keywords</i>	243
<i>Behave environment and project setup</i>	244
<i>Behave step definitions and implementation</i>	245
<i>Behave Test execution</i>	247
<i>Behave Data Parameters, Data Parameterization and Background</i>	248
Difference Between Pytest, Robot framework, Behave and Unittest	249
Conclusion	249
Key facts.....	250
Questions.....	251
10. Selenium Automation Framework – Part 2.....	253
Introduction.....	253
Structure	254
Objectives	254

Page Object Model (POM)	255
<i>Why Page Object Model?</i>	255
Implementing Page Object Model.....	256
<i>Advantages of Folder Structure.....</i>	257
<i>Steps for Page Object Model (POM).....</i>	257
<i>Best Practices for Implementing the Page Object Model.....</i>	267
Screenshots.....	268
PyAutoGUI	269
Configurations.....	271
<i>Configparser</i>	272
Logging.....	277
<i>Python logging.....</i>	278
<i>Logging level</i>	278
<i>Logging Formatter.....</i>	279
<i>Python logging in Framework</i>	281
Parallel Test Execution.....	283
<i>Pytest-xdist.....</i>	283
Data – Driven Testing with Data Source.....	284
<i>Openpyxl.....</i>	285
<i>CSV.....</i>	287
NumPy.....	291
Pickle.....	291
DateTime.....	293
Random	295
Faker.....	296
Conclusion	297
Key facts.....	298
Questions.....	298
11. Mobile Automation Framework.....	301
Introduction.....	301
Structure	302
Objectives	302
Allure reporting tool.....	303

<i>Installation</i>	303
<i>Decorators in Allure reporting</i>	304
<i>Screenshot with Allure Report</i>	305
<i>Executing test and opening Allure Report</i>	305
Designing Appium Framework.....	309
<i>Folder structure</i>	309
Implementation of Appium Framework.....	310
<i>Driver Initialization</i>	310
<i>Base Page and Screenshots</i>	311
<i>Configurations and Utilities</i>	315
<i>Pages</i>	317
<i>Tests and Execution</i>	320
<i>Execution of tests and output</i>	324
<i>Troubleshooting</i>	325
Conclusion	326
Key facts.....	326
Questions.....	327
12. Dockerized Selenium Grid	329
Introduction.....	329
Structure	330
Objectives	330
Virtualization	331
<i>Limitations of virtualization compared to containerization</i>	331
Docker	333
<i>Advantages of Docker</i>	334
<i>Docker use cases in software development</i>	334
<i>Docker terminology</i>	335
Docker installation.....	338
<i>Docker Mac installation</i>	338
Docker commands.....	341
<i>docker pull <image_name></i>	341
<i>docker build -t <image_name></i>	342
<i>docker run <image_name></i>	343

<i>docker stop <container_id></i>	343
<i>docker rm <container_id></i>	344
<i>docker images</i>	344
<i>docker push <image_name></i>	344
<i>Docker volume</i>	344
<i>Docker Compose</i>	345
Running Selenium Tests with Docker	347
Selenium Grid.....	351
<i>Advantages of Selenium Grid</i>	351
Selenium Grid with Docker.....	352
<i>Advantages of Selenium Grid with Docker</i>	353
<i>Running Selenium Grid in Docker</i>	354
<i>Executing parallel test</i>	358
<i>Scaling Number of Nodes</i>	362
Conclusion	363
Key facts.....	363
Questions.....	364
13. Bonus Chapter – Python Interview Questions.....	365
Introduction.....	365
Program 1	366
Program 2	367
Program 3	367
Program 4	370
Program 5	371
Program 6	371
Program 7	372
Program 8	374
Program 9	375
Program 10.....	376
Program 11.....	376
Program 12.....	377

CHAPTER 1

Testing Process and Role of Automation

Introduction

The definition of software testing is simple and has not changed since its origin. The actual developed software is in sync with the expected software, intended to be developed as defined in the business specification

The goal of software testing is to find potential errors in the developed software. Today, modern software testing is done using different automation tools, although a couple of years ago, the scripts were written manually and manual testers were doing the validations manually. Nonetheless, there remains a need for manual test engineers in high-sensitive software domains, for specific high-critical testing, which requires manual intervention.

Various questions might arise in our minds when we think of software testing. Where did it all start? How has software testing evolved? What is the standard testing process? What invoked the need for automation? In the field of software testing, knowing where and why it originated, will help understand the scenarios, different software testing types, their significance in the process, and how to implement them

during real-time projects.

In this chapter, we will go over some of the majorly used concepts in software testing. This will provide you with great skills and a strong grip on the core concepts, while performing testing in real-world projects, as well as provide you with a great amount of confidence facing the software testing interviews.

Structure

In this chapter, we will discuss the following topics:

- Significance of software testing
- Core concepts of software testing
- Types of software testing
- Software Development Life Cycle (SDLC)
- Defect/Bug life cycle
- Automation in software testing

Objectives

By the end of this chapter, the reader will be able to understand the importance of software testing in the software industry, along with core concepts and terminologies used in software testing, different testing types, and their definitions. The reader will also have a complete understanding of the process of software testing and the defect cycle within the software development process and models, followed by testing automation significance.

Significance of software testing

Software Testing is a mandatory step in the software lifecycle. The completion of this step provides confidence and guarantees that the product is of a standard to be pushed to production for customers. It is significant because these software applications are bound to have errors, and identifying them in the early stage saves a great sum of money and time.

There are so many incidents that occurred in the past, that were caused due to software glitches. One such incident is the multiple Boeing 747 Max crashes in recent

times. It was concluded that the key factor causing the crash was the **Maneuvering Characteristics Augmentation System (CAS)**. It is assumed that maybe something went wrong during the testing of the updated system. The tragedy could have been avoided with complete comprehensive testing; the testers do impact the world. No major product launch happens without testing; any product, or even an app on

the phone you are using right now, is delivered to you after testing. Testing is very crucial to identify any bugs or errors in the system early in the stage, so that they can be fixed before being delivered to customers. Moreover, apart from quality, this ensures dependability, security, and performance, which can benefit in cost and time saving, and thus customer satisfaction. As evident from the example of Boeing 747 Max mentioned previously, the company lost its reputation and dependability, the cost for the company in compensation, and the settlement was around 3-4 billion dollars.

A few major reasons that make software testing crucial in the software development process are as follows:

- Software testing indicates the bugs and defects in the product that may have occurred during the development phase.
- Software testing provides a smooth user experience for the customer. Thus, the company can gain their trust and confidence.
- Software testing ensures that the application's performance is intact for updates or the addition of new features to existing applications.
- The application's continuous software testing over time creates a platform for the developer to improve the development process. This prevents repeating the same error that occurred before, thereby reducing the coding cycles.
- Software testing makes the process cost-efficient, by capturing early defects.

Software testing is a continuous process of delivering a clean product. Here are some major benefits of testing for companies and customers:

- **Better business optimization by reducing cost:** It is very critical to the project, to figure out at which level of the software development, the bug was identified. The later the stage that the bug was raised, the more is the cost for the company.
- **Security:** The testing phase case significantly finds the vulnerabilities in the software, which can prevent hackers from hacking the system. This ensures that the customer data or any significant information is safe.
- **Performance and efficiency of the application:** This is closely related to the reputation of the company. During the testing phase, the performance of the application can be identified, which ensures that in the long run, the

or the application can be identified, which ensures that in the long run, the customers are satisfied.

- **Reputation:** For any industry, the consumers are the most important part. As consumers, we too depend on companies that produce reliable products. The basis for this comes from how much time is spent on testing the product, to a point of its result in customer satisfaction.

- **User satisfaction:** Customers are the highest priority for any industry. Customer satisfaction is directly proportional to the customer's flawless experience with the product, which is tied to how much testing was performed to match the customer's expectations.
- **Support to the development process:** Regression testing of a product and tracking the testing over time, helps developers consider these potential error scenarios in the upcoming development cycle.

Core concepts of software testing

Some of the core concepts of software testing are as follows:

Software testing, quality assurance, and quality control: The terms software testing, quality assurance, and quality control are used closely with each other. Although they are thought to be the same, there are subtle differences among them all and these terms serve different purposes. Software testing is a process used to find possible bugs, defects, correctness, completeness, and quality of the developed software. Software testing is the core of the testing process, as shown in *Figure 1.1*, and it finalizes the software application by closing the gap between developed software and the business requirement, before the software is released to production for customer use. *Figure 1.1* illustrates these concepts of software testing:

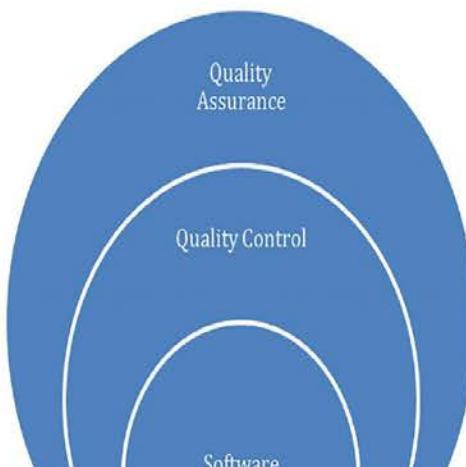




Figure 1.1: Concepts of software testing

Quality assurance is process-oriented, where testing and analysis are done to prevent error in the system before it happens. As shown in *Figure 1.1*, Quality Assurance is the outermost layer of testing which prevents the possibility of errors. This method provides assurance, that all the business requirements will be achieved or delivered as per the expectations of the consumer of the product. A high-quality software application cannot simply be achieved by testing the application. The defects occur because something somewhere did not work as expected; testing can find these defects. It will, however, not be able to prevent them from happening. Thus, the process that allowed these defects in the first place, needs to be identified and re-engineered.

Quality Assurance makes the development process more effective by taking into consideration the development techniques, pre-design, management, and project analysis, and so on. Quality Assurance comes under the category of Verification, which is a static analysis technique, where testing is done without executing the code.

Some examples of quality assurance in the software development process are as follows:

- Software development planning.
- The team resources are involved from the early stage of product development so that they are trained and have comprehensive knowledge of the application.
- Requirement documentation.
- Reviews and walkthroughs throughout the process.
- Internal and external audits.

Quality control is a Product-Oriented approach, that acts as a bridge between software testing and quality assurance. As shown in *Figure 1.1*, it lies in the middle, to determine if the application has any defects, as well as to make sure that the software meets all the business requirements. It also mainly focuses on the final product. Quality Control ensures that standards are followed, and is performed

after the software feature is developed, during the testing phase of the software development life cycle. It is also considered under the category of Validation, which is a dynamic analysis technique, where testing is done by executing the code.

Some examples of quality control in the software development process are as follows:

- Inspections and Peer Code Review
- Debugging Requirement documentation

Software Testing Process

The software testing process is as follows:

- **Test Strategy:** As the name suggests, a test strategy is a high-level approach to how testing of the application will be carried out in the given cycle. This is documented by considering the test environment, test data, tools, risk analysis entry criteria, exit criteria, and so on.
- **Test Plan:** This is a comprehensive document that lists all the activities to be performed in the testing. The test plan records requirements in scope, and testing is to be performed during different stages of development, customer sign-off, and so on.
- **Test Scenario:** It is the description of the use case, an outcome of business specification, and it determines all technical aspects and objectives.
- **Test Case:** This is a more specific step of testing linked to a given scenario. It comprises test steps, test data, the execution result of each step, and so on.
- **Test Execution:** The test cases are executed using specific test data to determine the application behaves as expected and meets the specified requirement. The goal is to detect defect or issues in the system that could impact application functionality, security, performance, or other quality attributes.
- **Test closure:** It is the final phase of the testing process ensuring testing objectives are complete and the application is reliable, stable, and ready for release to production. The phase involves analyzing test results, compiling test reports, and communicating the findings back to stakeholders.

Types of software testing

Let us look at the broad classification diagram shown in *Figure 1.2*, before deep diving into different types of testing. In a wide sense, the testing of an application

iving into different types of testing. In a wide sense, the testing of an application can be categorized as Manual and Automation testing. Manual testing can be further classified into Functional and Non-Functional testing, both of which have subcategories based on the definition:

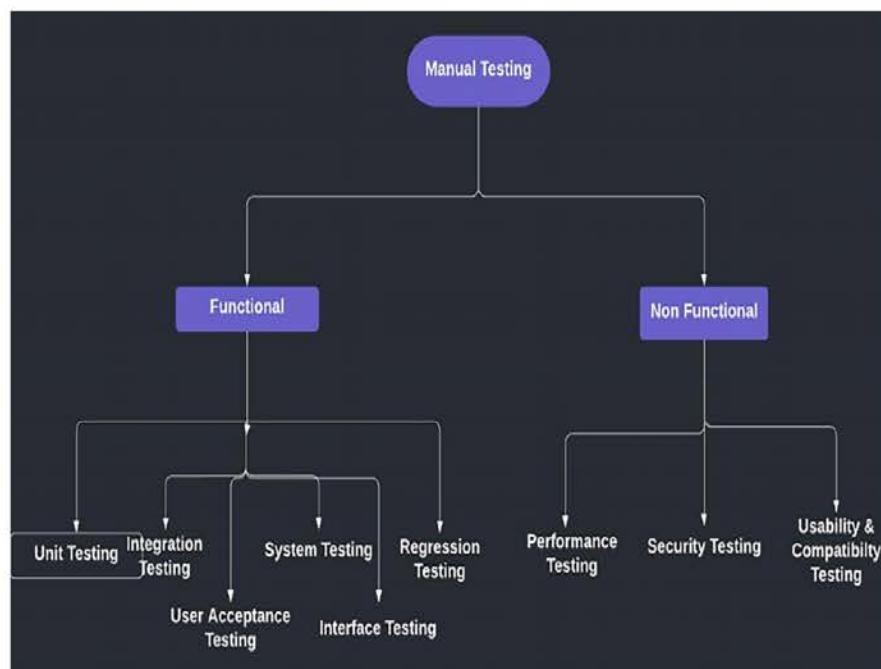


Figure 1.2: Manual Testing and its types

Let us now look at the definition of manual testing in detail.

Manual testing

This type of testing is done by manual testing engineers and performed manually by executing and validating the test cases interacting with the application. All the test cases and execution test results can be stored and tracked in a test case management tool or a spreadsheet.

Advantages of manual testing

The advantages of manual testing are as follows:

- Mimicking the customer experience is only possible via Manual testing.
- It is comparatively cheaper than Automation testing and requires less maintenance.
- Highly efficient in low-critical applications, where investing time and money in automation does not provide business value.
- Best testing solution during early stages of application development.
- Production day release testing for new features is more efficient with manual testing.

Disadvantages of manual testing

The disadvantages of manual testing are as follows:

- Considered risky and highly susceptible to human error.
- Inefficient and unreliable when a huge number of test cases validation is required during pre-production release and time-bound testing.
- Logging, reporting, and documenting execution results is extremely time-consuming.
- Performance testing of the application is not possible or inaccurate with manual testing.

Functional and non-functional testing

When testing is done considering the business requirement of the application, it is called functional testing. This is a type of black box testing where only the output is validated against expected results that are based on the given input. Here, the internal flow or program structure is not considered.

On the other hand, Non-Functional testing assesses the application behavior which is not critical to the functionality of the application but contributes to the consumer experience.

Non-Functional testing includes Penetration testing, where the application is tested for security vulnerabilities, Performance testing, Usability, and Compatibility testing, where the application's cross-platform support, Exploratory testing, and Accessibility testing are performed.

Unit testing

When testing is done to validate the smallest module or block of code of the application, which could be a function or a procedure, it is called Unit testing. Unit testing of the application is performed during its development, by developers or by testers. Unit testing is done by passing an argument to a function, and validating that the expected value is returned or executed by the function. The benefit of Unit testing is that it divides the whole application into the smallest of pieces, and validates its correctness, thereby helping to identify and fix an error in the very early stages.

White-Box, Black-Box, and Grey-Box Testing

White-Box is a type of testing that is performed considering the inner component of the application. On the other hand, Black-Box is a type of testing that, like Functional testing, is performed considering only the expected result, based on provided input, and not the internal structure of the application. The grey-box testing is a software testing type that is used to test the behavior of the application and the internal component of the application.

Integration testing

This type of testing validates whether different modules that make the application, are working well together. This can also be done to validate two applications working together as expected.

User acceptance testing

Also called 'end user testing' or 'application testing', acceptance testing is performed by testers or consumers in the production environment. This testing output determines if the application or feature is ready to be released.

Alpha and beta testing

Both Alpha and Beta testing are kinds of User Acceptance testing, where Alpha testing is done before the release in order to find bugs in the application, and beta testing is performed by real users in the real environment.

System testing

In System testing, all the components of the application are tested as a single unit. This ensures that the developed software meets the business requirements. It is high-level testing performed after Unit and Integration testing.

End-to-end testing

This involves the complete testing of the application's workflow, from start to finish in a real-world scenario, to validate the application for integration and data integrity.

Sanity testing

It is a subset of regression testing, performed on the application to ensure that the newly added feature or bug fixes have not broken any existing functionality.

Smoke testing

It is the testing performed to make sure that the core functionalities of the application are intact.

Software Development Life Cycle (SDLC)

Software testing is one of the phases in the development process. Software testing should happen in sync with the model followed for development. Let us discuss the flow of testing the application. There are multiple models for software testing, and the predominantly used one in the modern industry is the Agile methodology. Even though there are many models such as Agile, Spiral, V-Model, Rapid action, Big Bang, Waterfall, and so on, we will look at 3 famous models including Agile, to see how the models evolved.

Waterfall model

A majorly used model around 10 years ago, the Waterfall model is the earliest **Software Development Life Cycle (SDLC)** used in software development. It follows a distinctive sequential approach to performing Software testing. Software testers perform one step at a time, in each of the software development phases, starting from documenting the requirements, to maintenance after deployment. The next phase can only begin when the previous phase is completed, and hence it is also called the Linear-Sequential life cycle model. Refer to *Figure 1.3* for an illustration of the Waterfall model:

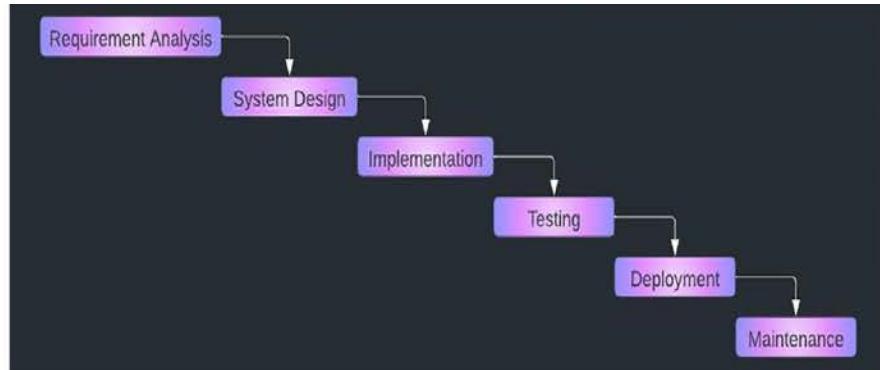


Figure 1.3: Waterfall Model

As shown in *Figure 1.3*, the waterfall model comprises of the following broadly classified phases:

- Requirement gathering and analysis is the first step in any software development process, where the business requirement and specifications

are gathered and documented. The testers understand the requirements and come up with a testing strategy.

- In the System design phase, the software and hardware requirements along with the development approach are finalized. This phase is where the testing team is finalizing the testing requirements.
- This is followed by the actual implementation of the logic in the Implementation phase.
- Then, all the required testing of the logic including integration (the link between two systems in software) and end-to-end testing (complete flow of a scenario), are performed. This is the phase where defects are logged and fixed, and there is continuous coordination between testers and developers.
- Upon completion of testing and fixing all the defects, the logic is deployed to production in the deployment phase.
- This is followed by maintenance of the deployed version and supporting customers in the last phase.

Waterfall model application

There is no ambiguity in this model. Neither is there going to be any requirement changes during the mid-phases. Moreover, the business specifications are clear and well-documented. There are no pending items from the previous phase, and there exists a clear understanding of the product development among the resources. This model is ideal for short projects with a small set of requirements.

Advantages of the Waterfall model

The advantages of the Waterfall model are as follows:

- The simple and linear approach of this model is easy to understand and use.
- It is also easy to manage since the deliverables in each phase are clear among the team.

Disadvantages of the waterfall model

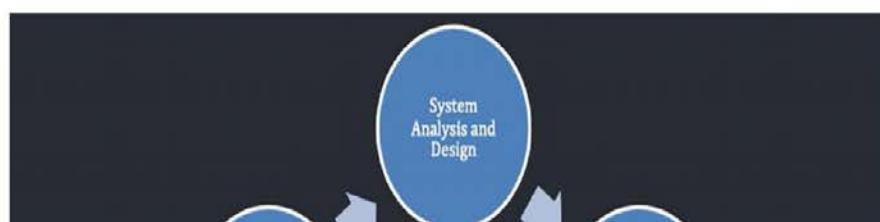
The disadvantages of the Waterfall model are as follows:

- The major disadvantage of the model is that it is time-consuming. The entire development process of navigating through each phase can take months and no working software is available until a later phase.
- Until the development process is going on, the testing team is waiting idle.

- If there is a change/update to the requirements, it increases uncertainty. Hence, in this model, significant changes to the business specification cannot be made.
- Within each phase, the progress is not measurable.
- The model can end up with a pile of errors in the testing phase because the integration and error risks are not considered due to integrations.

Iterative (and incremental) model

This is a model which was developed to address the shortcomings of the waterfall model. In this model, the software development process starts with the initial planning of the complete requirement in an Iterative (repetitive) and Incremental (Occurring in a short period) manner, with a new subset of complete requirements from the initial planning. This allows developers and the software development process to accept changes and improvements and can also accommodate changes to the requirements. The Iterative and Incremental Model is depicted in the following Figure 1.4:



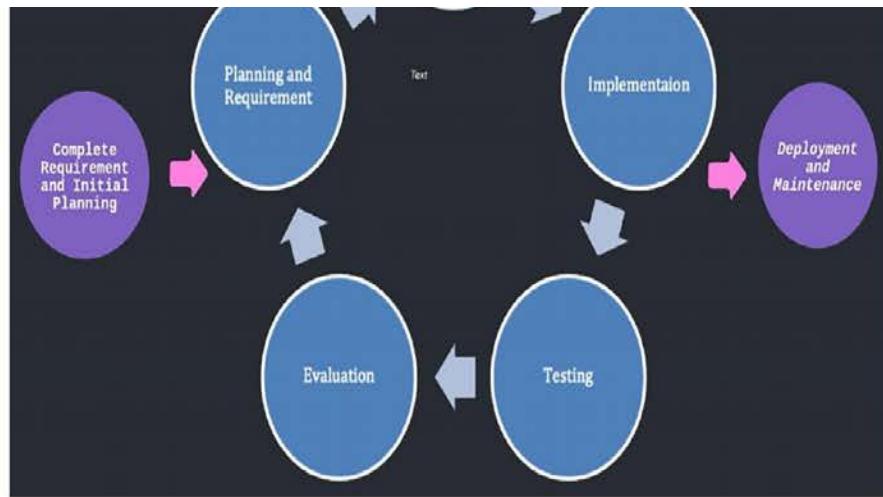


Figure 1.4: Iterative and incremental Model

Iterative and incremental model application

While the major requirements are outlined and planned, minor changes to the specification and improvements can be added during the process. Testing is done

in each iteration, giving confidence in the new addition. It accommodates the Integration testing during the iterations.

Advantages of the Iterative and Incremental Model

The advantages of the Iterative and Incremental model are as follows:

- A working model of the application in the early phase of development allows early feedback and enhancements.
- Testing is done in each iteration, resulting in the early detection of errors.
- Changes to software, hardware, and system design requirements can also be updated during the process along with business specifications.
- Within each iteration, the progress is measurable and can be improved upon.
- It is easier to prioritize the requirements and manage risk, where high-risk or high-priority requirements can be taken in early iterations.
- There is learning in each iteration that can be applied to the next iteration, which increases the efficiency of the development process.
- It has a lower risk in comparison to Waterfall Model.

Disadvantages of the iterative and incremental model

The disadvantages of the iterative and incremental model are as follows:

- The model is still not susceptible to major changes or improvements to the application which can incur higher costs.

requirements, which can incur higher costs.

- The number of Iterations depends on several factors; thus, the model does not provide enough control over the time of completion.
- Not ideal for smaller projects.

Agile methodology – SCRUM

Agile is the most widely used and accepted software development model. This again relies on an Iterative and incremental model, with a focus on deliverability and adaptability with more control over time, thus leading to customer satisfaction. In Agile, instead of developing the software iteratively and deploying at the end of all iterations, the product is broken down into incremental builds. Hence, each iteration has a product deployment. Agile teams are designed to be able to change, improve, stabilize, and recover faster in case of a failure. In Agile, a team can prioritize the requirements and adapt in real-time, and thus Agile teams deliver.

Let us discuss Scrum Agile briefly. Scrum is the most widely accepted framework of Agile, which has stages of development called Sprints. Each sprint is of 2-4 weeks, although a week does not necessarily start on a Monday. In a real project, every

day starts with a small meeting called daily scrum managed by the scrum master for a progress update, blockers, and resolving issues to plan a productive working day. Usually, all the teams in a company have an all-day meeting, once a quarter, to plan out each team's goal, and dependencies called epics. Epics will have sub-stories and tasks that are required to be achieved. The refinement meeting helps the team pick up the Sub stories that are in individual sprints, considering the efforts and complexity. The story points are given to these stories which determine the time it could take to complete. If the story points cross certain limit, the story is broken down into two sub-stories which can be completed in two sprints. The beginning of a sprint starts with sprint planning to set the expectation. Here, each team member picks up these stories based on bandwidth, which is called team capacity and is calculated based on people's availability in each sprint. At the end of the sprint, there is a Sprint review meeting to look at the finished work and get feedback from stakeholders and customers, based on which product owner can add, remove, or update the backlog items. At the end of each sprint, there is also a Retrospective meeting where the team looks back at the last sprint and opens a platform to answer questions and comments on what went well, which team should continue to work, what could be improved upon, appreciation of team member and so on. This helps the team to keep track of the good and the bad, and thus improve from there.

SCRUM Agile methodology application

At every stage in Agile, there is a demo of a piece of working software, that becomes a communication with clients and input for requirements, making the clients part of the development process. Agile methodology is susceptible to changes.

Advantages of SCRUM Agile methodology

The advantages of SCRUM Agile methodology are as follows:

- A 2 to 4 week goal allows the team members to stay committed and motivated.
- Daily scrum meetings allow the team to identify blockers and resolve them early.
- Agile teams are designed to absorb unexpected errors and handle them.
- The definition of done in each story helps the developer stay clear on requirements.

Disadvantages of SCRUM Agile Methodology

The disadvantage of SCRUM Agile Methodology is as follows:

- A well-organized team is a necessity to sync on priorities, team sprint capacity, and communication channels with customers and within team

members. Any disruption can cause burnout in the team. The Scrum Master plays a key role in this.

Defect/Bug Life Cycle

The main purpose of Software testing is to find errors in the application. Testers use terms such as bugs, defects, errors, faults, and failures whenever the application is abnormal. A Bug is an error detected during the testing stage, that occurs due to an error in the code at a single unit level. A defect occurs when there is a variation in business requirements after production release. The testers or business analysts, based on input from customers, raise defects and based on priority defects, they are classified as High, Medium, and Low. Based on severity, defect types are critical, major, moderate, minor, and cosmetic. An Error is a developer's deviation due to an unclear description of the specification.

The defect life cycle tracks the life of an error from raised by testers to fixed, validated, and closed.

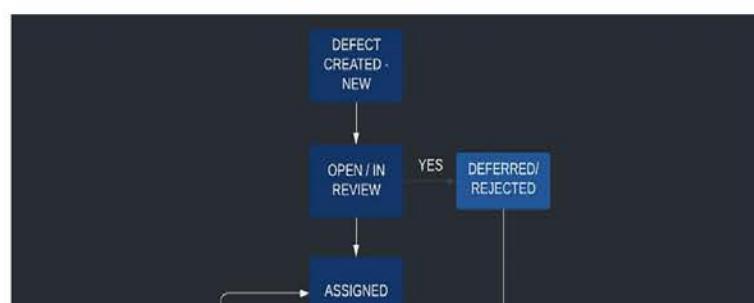
Different states of defect

The different states of Defect are as follows:

- New:** When a bug is discovered in the testing stage, or a new defect is created and unassigned from production.
- Open/in review:** The main responsibility of the tester creating the defect is to analyze the severity of the defect and bring it up for discussion with the team, if required. This will help the team to prioritize the issue. In a Sprint agile team, this will be refined, assigned story points, and then assigned to a developer, to work on a fix. QA is identified for testing, depending on team capacity and priority.
- Deferred/rejected:** There are a few reasons why a bug created is rejected, deferred, or closed immediately. Firstly, the defect is not reproducible because of some network or data issue, where it could also be intermittent, and it is the tester's responsibility to test it thoroughly. Secondly, the defect is expected behavior as per the specification; bring it to the product owner to confirm, or if the defect cannot be fixed.
- Assigned:** The Developer is identity-based and will be picking up the story to work on the fix in the same sprint or the next, based on the severity and priority of the bug.
- In progress:** When the developer assigned has started working on the fix.

- Code review:** After the developer's fix is in place and validated, send the code for peer review.
- Deployed and ready for testing:** After code review, the code is deployed to the test environment and ready for testing for the QA member to pick up.
- QA testing:** When the defect is under testing and the QA member is executing the test case for the defect and validates all potential scenarios.
- Done:** This denotes that the defect is completely tested and validated and is ready to be deployed to production.

Figure 1.5 illustrates the different states of defect:



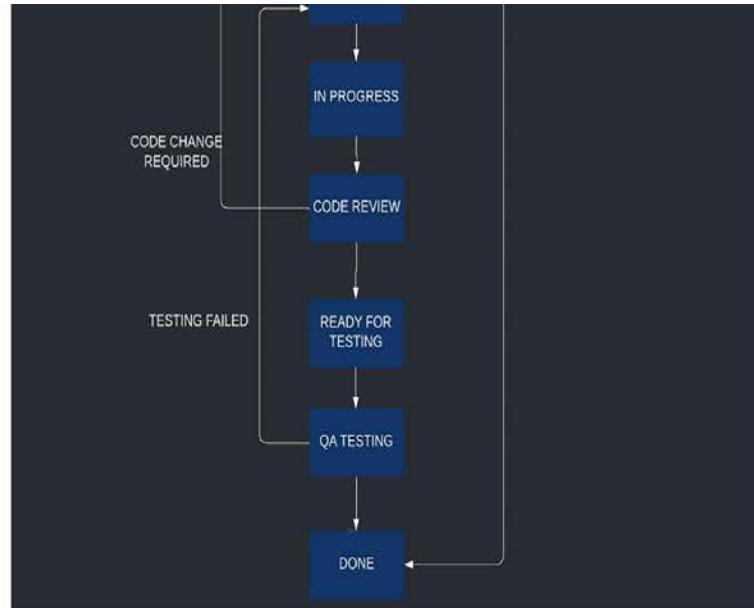


Figure 1.5: States of defect

Automation in software testing

We can define Automation testing as a process of running or executing test cases using tools, scripts, or software. No developed software is deployed to production without testing, because even after multiple testing cycles from unit testing to UAT,

no product is completely free of errors. In a fast-paced software market, where there are continuous deployments to meet the high demands and keep up to completion, Automation Testing provides benefits and ways to increase test coverage, efficiency, and confidence over Manual testing.

Even though it is evident, let us look at some of the differences between manual testing and Automation testing. Manual testing is prone to human errors and is also time-consuming. Hence, it is inefficient and unreliable. Compared to manual testing, Automation testing is more reliable and faster, as tools and software are performing the test. Automation testing is applied when tests are required to run multiple times, such as regression and performance testing. Manual testing is ideal for a test case with high-security risks.

Advantages of automation testing

The advantages of automation testing are as follows:

- **Low cost:** Automation brings down the possibility of error. Hence, the cost of manual resources is low.
- **Time efficient:** Automation scripts once created can be run multiple times.

- **Time efficient:** Automation scripts once created can be run multiple times over and over with a click of a button. While manual testing could take days for a given testing task, automation can do it in hours.
- **Reliable:** Manual testing is prone to human errors, but automation does it precisely without any assumptions and provides reports of every step.
- **Increases test coverage:** Automation testing can increase the testing scope by validating the sanity, smoke, and regression every day. It makes sure that any new change is not breaking the application to testing complete regression and lengthy end-to-end testing, during every release.
- **Adds to team confidence:** Looking at the automation test results and detailed report provides more confidence to the team compared to manual testing validation.

Conclusion

Software testing plays a significant role in the process of software development. The understanding of software testing planning, application of different types of testing in the process, a core understanding of testing and defects cycle with comprehensive knowledge of the importance of automation to improve the efficiency of testing, are the skills in most demand in the current software market. The understanding and ability to work in an agile environment will greatly improve productivity.

With the understanding of core concepts of Software testing, let us jump to the next steps in software testing automation with Python. We will go over the core concepts

of Python, which is a very popular and powerful programming language, and we will also learn how to leverage Python for automation.

Key facts

- Software testing's main goal is to find bugs in the application. It gives a great level of confidence before the software is made available for the customers, and ensures smoother, high-performance, efficient applications for users.
- Software testing is broadly classified as Functional and Non-Functional testing. Functional testing is derived from the business requirement and Non-Functional testing contributes to better the behavior of the application.
- There are different models of the **Software Development Life Cycle (SDLC)**, the most in-demand being SCRUM Agile methodology. This makes a robust development and testing team. The details are mostly asked in interviews.
- The significance of automation in the software testing process improves

the efficiency, and productivity of the testing process and thereby of the development and deployment process.

Questions

1. What is Software Testing and what is its significance in the Software development process?
2. What are the differences among software testing, Quality control, and Quality assurance?
3. What is the difference between Functional and Non-Functional Testing?
4. What is the Agile Methodology process?
5. What are the different phases of the Defect life cycle?
6. What are the advantages of Automation Testing over Manual Testing?
7. What is the difference between Sanity and Smoke testing?
8. What is the difference between verification and validation?

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



CHAPTER 2

Python Programming

- Setup and Core

Concepts

Introduction

Python is the most popular and widely used general-purpose, high-level object-oriented programming language, designed by Guido van Rossum and was released in 1991. In the modern software industry, Python is acclaimed for its enormous flexibility, readability, and adaptability features, present because of its simplified syntax and ease of learning. Python now has a vast community, support, and regular latest updates.

Python is platform-independent and supports Mac, Windows, Linux, and various other operating systems. Python is versatile and applied in major programming use cases such as web applications, API development, Machine learning, Data Science, Big Data, Database, Artificial Intelligence, and so on.

Python in Automation Testing, as well as the vast set of benefits and libraries that Python brings into testing, makes Python an extremely software automation testing friendly programming language. Its simple nature helps manual engineers migrate to automation and accomplish complex testing solutions.

Structure

In this chapter, we will discuss the following topics:

- Advantages of Python
- Getting started with Python
- The Python Interpreter
- Executing the first Python Code
- Virtual environment and Requirement.txt file
- Core concepts of Python
 - Variables and data types
 - Types of operators in Python
 - Lists in Python
 - Tuples
 - Sets
 - Dictionaries
 - Conditional statements
 - Loops in Python
- Functions in Python
- Modules, packages, exception handling in Python
- Classes and Objects in Python
- Inheritance in Python

Objectives

This chapter will help us understand the basics of Python, from installation, setting up environment variables, **requirement.txt** file, to the concepts of code structure in Python with execution flow, writing Python programs, handling different variables, operators, expressions, and data types including Strings, and so on. We will also learn about program flow control using **if-else**, **for** and **while** loop, data structures with lists, Tuples, as well as Dictionaries, Sets, and Functions. This will refresh the knowledge or help those who are learning Python from root, or migrating from other programming languages.

This chapter aims to provide the Python overview, so that it becomes easy to understand when concepts are applied with Selenium web driver for automation testing.

Advantages of Python

There are various advantages of using Python and they are as follows:

- **Ease of learning:** Python is more readable, has simple syntax, and is less complex, making it easier for beginners to learn and migrate.
- **Multi-platform support:** Python can run on multiple platforms such as Mac, Windows, Linux, Solaris, CentOS, Ubuntu, and so on, without having to make changes to the code. Thus, applications written on one platform can be easily executed on another.
- **Strong active community:** Python has an active community and forum to provide support and software solutions, to help with programming errors and issues.
- **Libraries Support:** The inbuild Python libraries collections are huge. For external libraries, the Python package manager (pip) allows the import of packages from the **Python Package Index (PyPi)**.
- **Open-source:** Python is under OSI and FSF-approved open-source programming language licensed under **General Public License (GPL)**. Hence, it is free to download, use and distribute.
- **Productivity:** Due to its simple and compact syntax, developers spend less time writing the code and more time providing software solutions instead.
- **Testing:** Along with Selenium support, Python provides an inbuild testing framework for debugging and fast workflows such as pytest and splinter.
- **Dynamic in Nature:** The programmer does not have to declare the variable data types in the code, assigned automatically during run time.
- **Interpreted Language:** A program written in Python language is executed the instructions line by line via an interpreter.

Getting started with Python

The first step in Python is installing it in the system. Visit <https://www.python.org/downloads> to download the latest or older version of Python as per requirement. Python 3.11.2 version is the latest and is currently in development.

Installation of Python in Windows

To download Python in Windows, follow the given steps:

1. Click on **Download Python 3.10.7**, as this is the latest version. Refer to *Figure 2.1*:

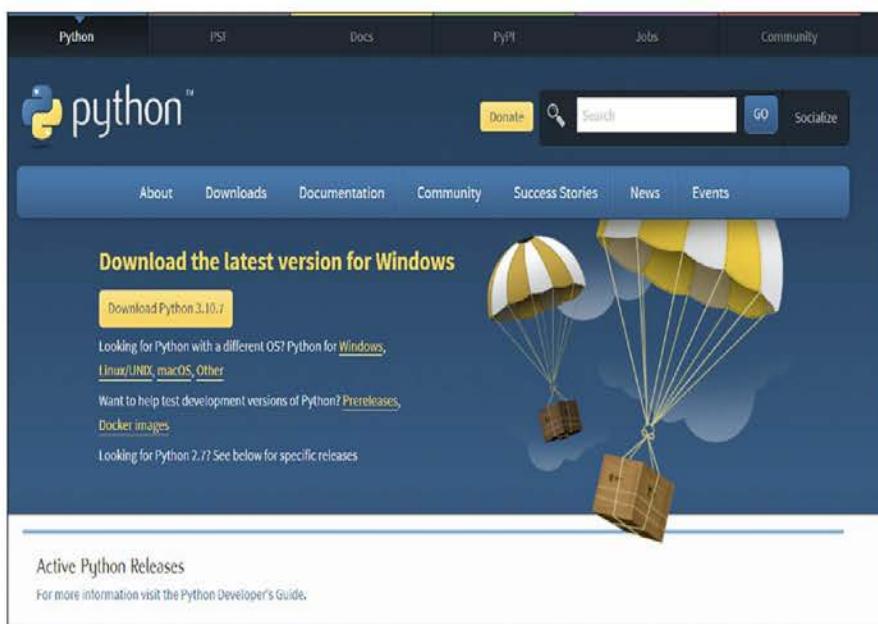


Figure 2.1: Python Download Website

2. Open the downloaded .exe file to start the setup process. Check **Add Python 3.10 to PATH** checkbox, as shown in *Figure 2.2*, and click on **Customize installation**:



Figure 2.2: Python Windows installer

3. Add optional features, and advanced options and customize the installation path directly to C drive as shown in *Figure 2.3* and *Figure 2.4*:

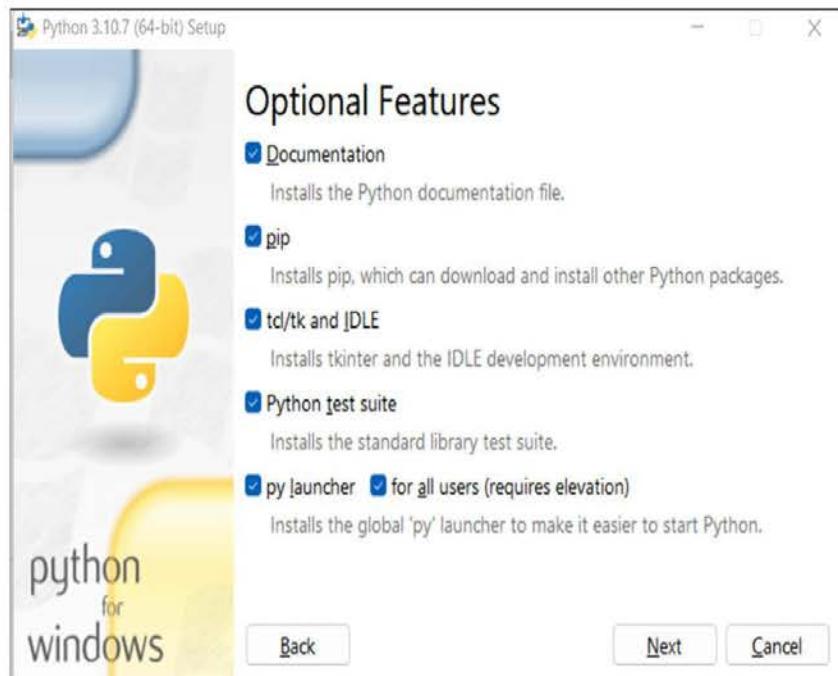


Figure 2.3: Python Customize installer – Optional Features

Refer to the following *Figure 2.4*:

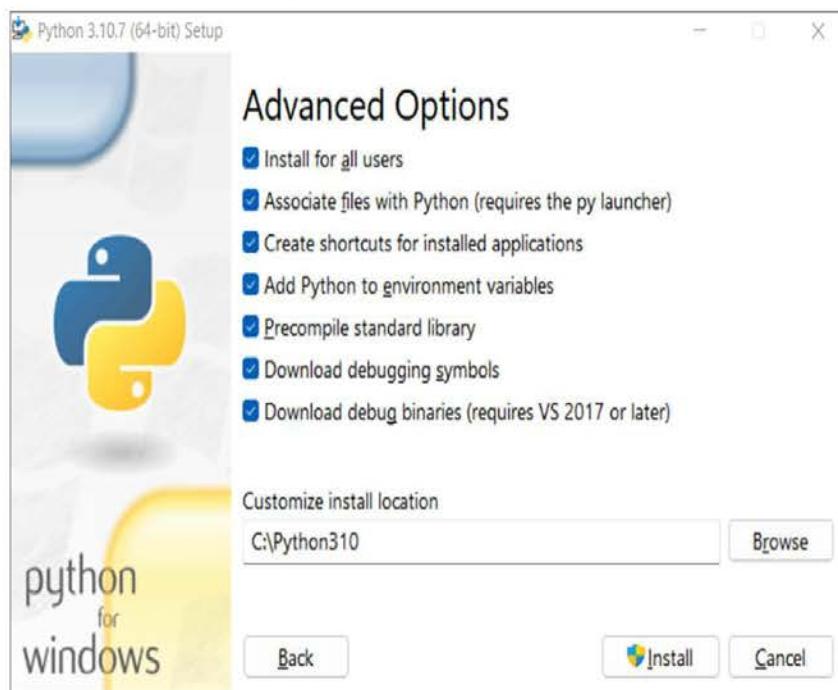


Figure 2.4: Python Customize installer – Advanced Options

24 ■ Selenium and Appium with Python

4. Installation should begin and should be successful, as shown in *Figure 2.5*:

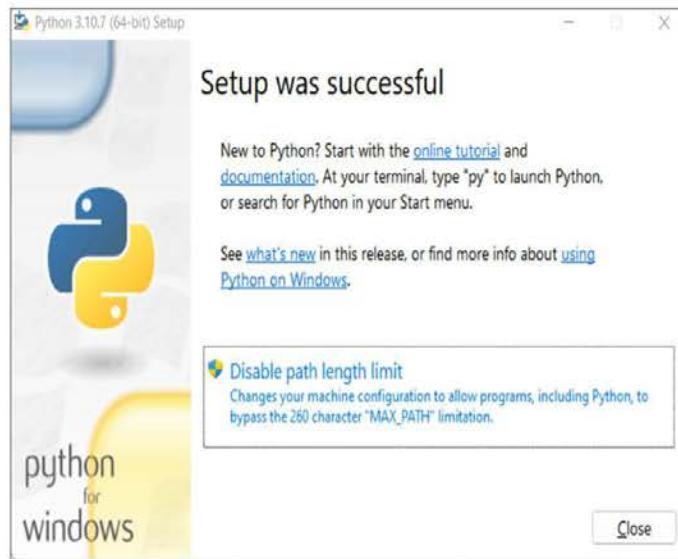
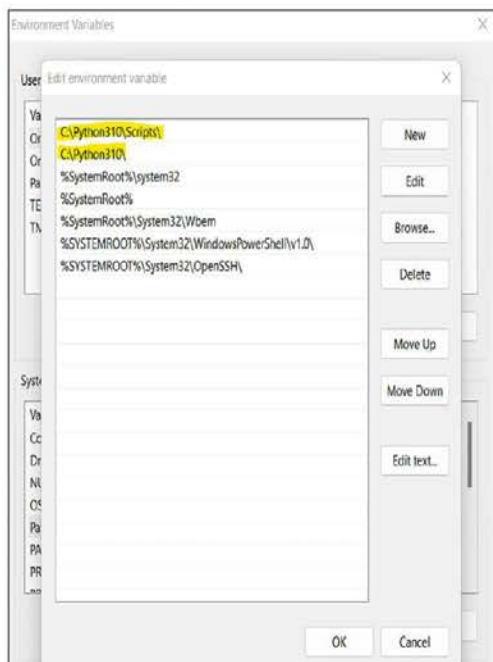


Figure 2.5: Python Installation Successful

5. After installation, make sure the environment variable is set up by the installer as in *Figure 2.6*. To navigate global, search Environment variable or navigate to the **Control Panel | System and Security | System | Advanced system settings**. Click on path in system variables.



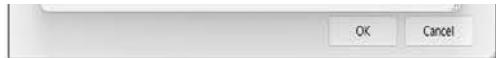


Figure 2.6: Python System path variable

Here are some Python environment variables to note:

- **PYTHONPATH:** This variable's function is like PATH system variable, as shown in *Figure 2.6*. It tells the interpreter location of module files that are written inside the program. This variable should include the Python source code library directory where Python is installed in the machine.
- **PYTHONHOME:** This variable is used to set additional directories where Python will look for modules and packages.

Installation of Python in Mac OS

To download Python on Mac OS, follow the given steps:

1. Click on **Download Python 3.10.7**; this is the latest version, as shown in *Figure 2.7*:

The screenshot shows the Python.org download page. At the top, there's a large yellow button labeled "Download Python 3.10.7". Below it, there's a section for macOS users with links for "Windows", "Linux/UNIX", "macOS", and "Other". There's also a link for "Want to help test development versions of Python? [Prereleases](#), [Docker images](#)". A cartoon illustration of two boxes descending from the sky on parachutes is visible. Below this, a table lists the active Python releases:

Python version	Maintenance status	First released	End of support	Release schedule
3.10	bugfix	2021-10-04	2026-10	PEP 619
3.9	security	2020-10-05	2025-10	PEP 596
3.8	security	2019-10-14	2024-10	PEP 569
3.7	security	2018-06-27	2023-06-27	PEP 537
2.7	end-of-life	2020-07-03	2020-01-01	PEP 373

Figure 2.7: Python Download Website

2. Once the `pkg` file is downloaded, follow the installation steps, as shown in *Figure 2.8*:

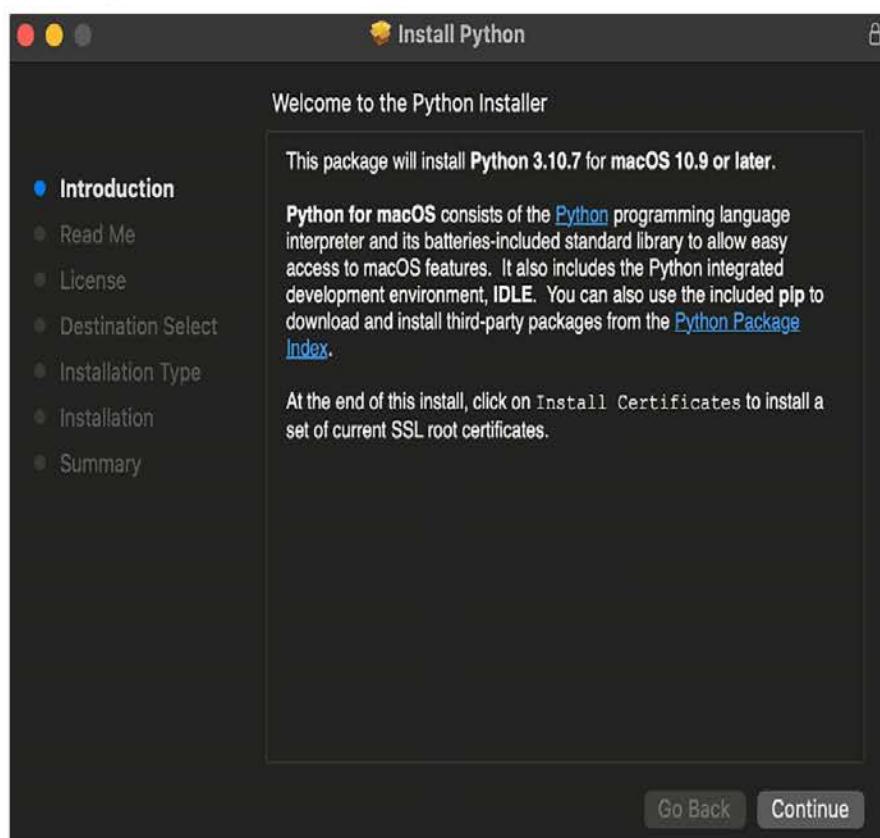


Figure 2.8: Python Installation window steps

3. Validate the successful installation by navigating to Terminal. Type `python3 --version`. You should see the latest version of Python. On successful installation of Python, you will be presented with the window shown in *Figure 2.9*:

A screenshot of a terminal window titled "yogashivamathivanan — zsh — 80x24". The window shows the user's last login information: "Last login: Sat Sep 10 13:47:17 on ttys000". Then it shows the command "python3 --version" being run and its output: "Python 3.10.7". The prompt "% " is visible at the end of the line.

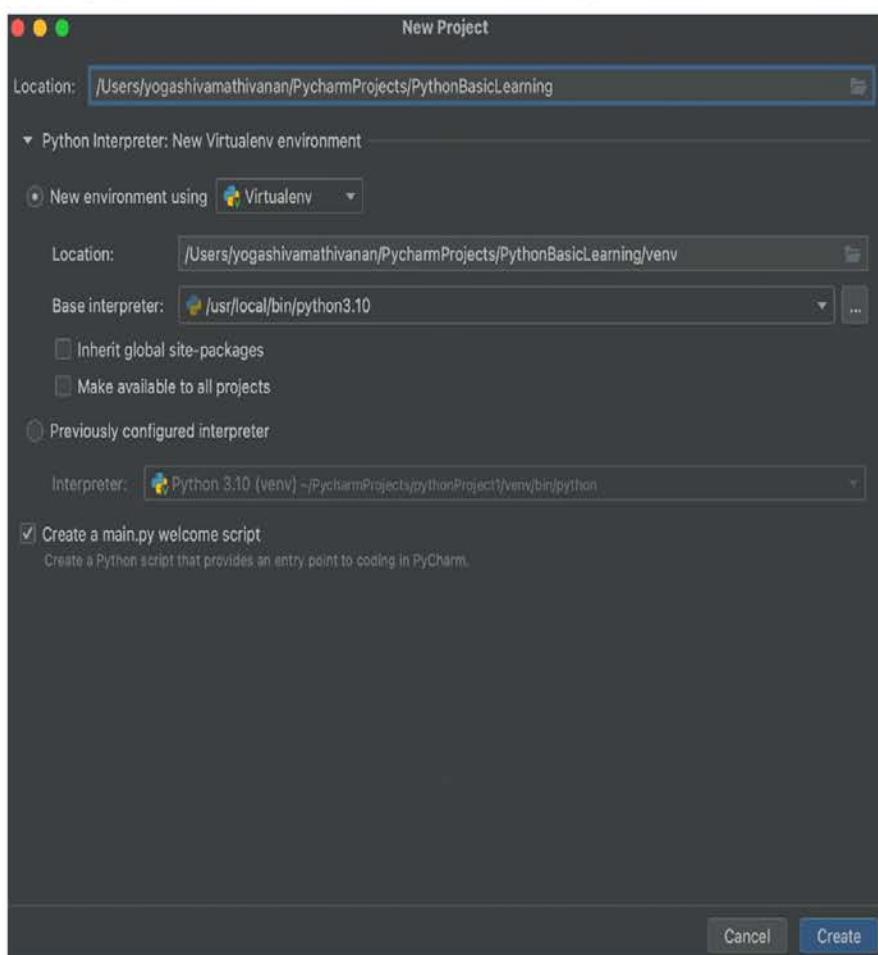


Figure 2.9: Python Successfully Installed

Installation of PyCharm

PyCharm is one of the most widely used **Integrated Development Environments (IDE)** by Professional Python developers and programmers. To install PyCharm, follow the given steps:

1. Click on the **Download Python 3.10.7**; this is the latest version. Navigate to <https://www.jetbrains.com/pycharm/download/> and download the Open-source Community version of the tool.
2. Once installed, open PyCharm CE and create a new project. Let us give the project name **PythonBasicLearning**. Refer to *Figure 2.10*:



The Python interpreter

Let us understand how any computer runs a program. The two basic components involved here are the processor, that is the **Central Processing Unit (CPU)** and the computer memory. The processor executes code stored in the memory, in the form of instructions called Machine Code and the result is stored back in the memory. When we write a source code, the language needs to be translated into machine code for a platform (such as Mac, Linux, Windows) that the processor can understand. The translator used here is called Compiler.

In Python, the flow is different. When we write a source code in Python and execute the program, the Python interpreter comes into the picture. The Python interpreter has two components: **Python Virtual Machine (PVM)** and a Compiler. When we execute the source code, the Compiler will convert the source code into a byte code (also binary 1's and 0's) instead of machine code. We use the concept of byte code in Python to achieve portability, which makes the Python platform independent. The PVM will read and execute the byte code on the hardware. Therefore, Python is both Compiled and interpreted language.

In *Figure 2.11*, the Base Interpreter is the Python version you want to use, to execute the Python file:

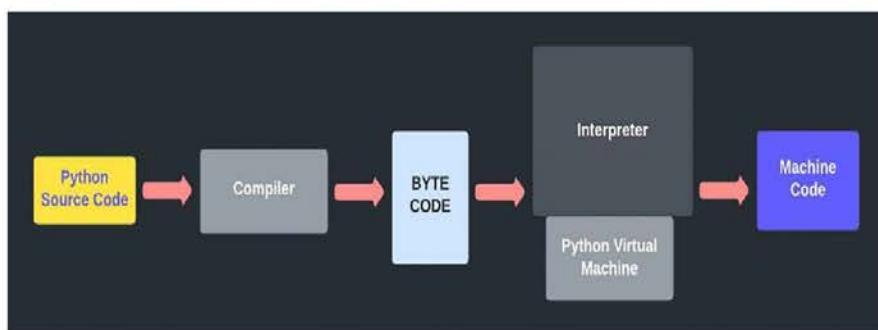


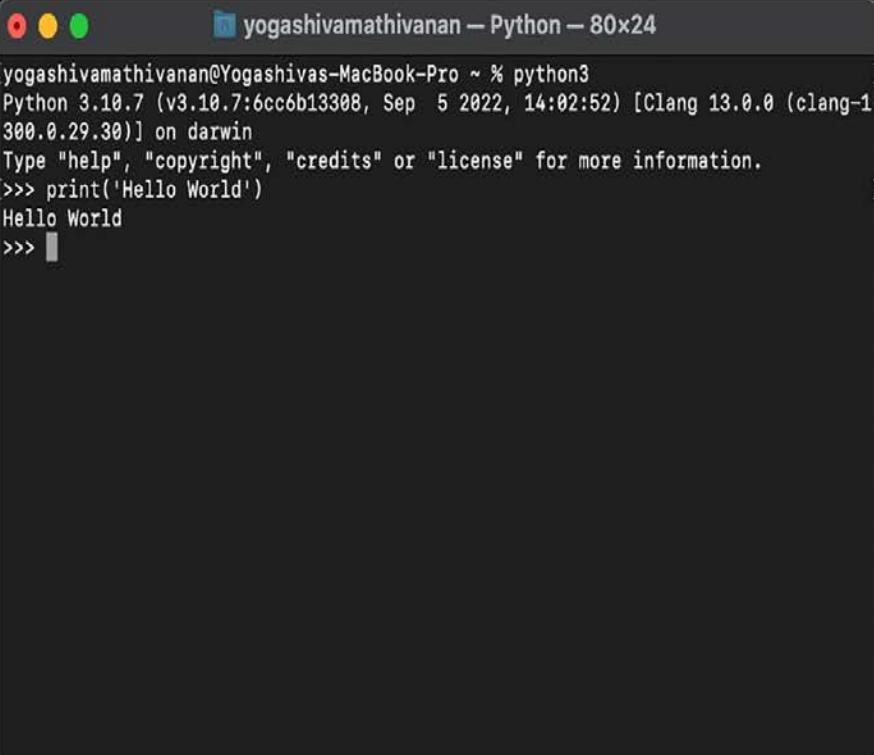
Figure 2.11: Python Program Execution Flow

Executing the first Python code

Now that we have installed the prerequisites, let us run the first Python code. We can run the Python code in one of the three ways mentioned as follows.

Using an Interactive Interpreter

Open Terminal and type `python3`. This will start the Python interpreter and allow us to run Python commands. Refer to *Figure 2.12*:

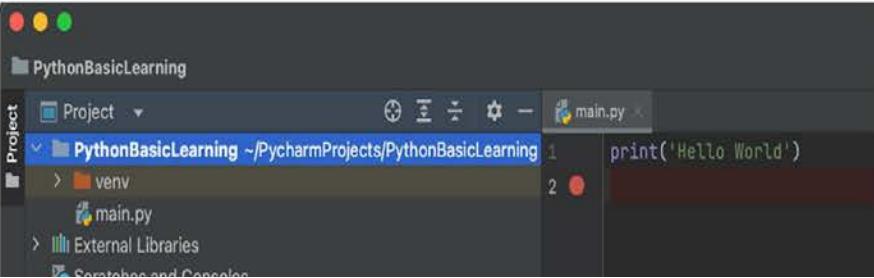


```
yogashivamathivanan@Yogashivas-MacBook-Pro ~ % python3
Python 3.10.7 (v3.10.7:6cc6b13308, Sep  5 2022, 14:02:52) [Clang 13.0.0 (clang-1
300.0.29.30)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> print('Hello World')
Hello World
>>>
```

Figure 2.12: Python Execution using Interactive Interpreter

Using a command line

The Python script is saved with a `.py` extension. We can invoke the interpreter to execute the script. Refer to *Figure 2.13*:



The screenshot shows the PyCharm IDE interface. The project navigation bar at the top displays the path: `PythonBasicLearning` → `Project` → `PythonBasicLearning` (~/PycharmProjects/PythonBasicLearning). Below the navigation bar, the project structure pane shows a folder named `venv` and a file named `main.py`. The code editor pane contains the following Python code:

```
1 print('Hello World')
```

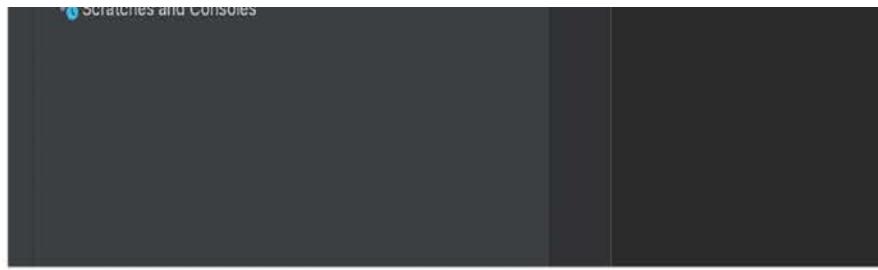


Figure 2.13: Python Execution using Command Line main.py file

The file **main.py** in *Figure 2.13* is executed in *Figure 2.14* using command line:

```
Last login: Sat Sep 10 14:23:39 on ttys001
yogashivamathivanan@Yogashivas-MacBook-Pro PythonBasicLearning % python3 main.py

Hello World
yogashivamathivanan@Yogashivas-MacBook-Pro PythonBasicLearning %
```

Figure 2.14: Python Execution using Command Line

Using an IDE: PyCharm

Once the New project is created, the PyCharm creates a default project structure with the **main.py**. Press the Green button on the top right or the **Ctrl + R** key. It will return the result **Hi, PyCharm**, as shown in *Figure 2.15*:

```
Pycharm Learning - main.py
Project: Pycharm Learning
In Pycharm Learning: C:\Users\yogashiva\PycharmProjects\Pycharm Learning\src\main\py
External Libraries
Scratches and Consoles

# Press Alt+F10 to invoke PyCharm 2020.1 Help
# Press Ctrl+F1 to search everywhere for classes, files, tool windows, actions, and settings.

def print_hi(name):
    # Use a breakpoint in the code line below to debug your script.
    print(f'Hi, {name}')  # Press F11 to step into code

# Press the green button in the toolbar to run the script.
# Press Ctrl+F8 to toggle the breakpoint.

if __name__ == '__main__':
    print_hi('PyCharm')

# See Python data at: https://www.jetbrains.com/help/pycharm/
```

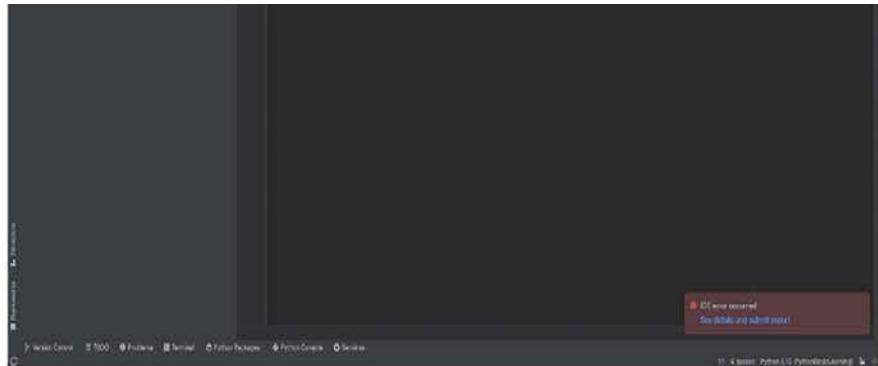


Figure 2.15: PyCharm New Project

Understanding `__name__ = "__main__"`

This is an early concept but when Python executes the file, Python sets a few special variables and `__name__` is one of them. When Python runs a Python file directly, it sets the variable value to `__main__`. When we import the first file into the second file and execute the second file, the variable `__name__` value of the first file is set to the name of the file name. As in *Figure 2.9*, the statement “`if __name__ == '__main__'`” is useful because it tells if the `main.py` file is run directly by Python or if it is being imported.

Virtual environment and Requirement.txt file

A virtual environment in Python is a self-contained directory that contains a specific version of the Python interpreter, along with its own set of installed packages. The virtual environment is used to isolate the project dependencies from system-wide Python installation and from other projects, allowing us to avoid conflicts between packages and ensuring that the project runs correctly.

To create a virtual environment, we can use the '`venv`' module that comes with Python 3. To create a virtual environment named '`myenv`', type the command "`python -m venv myenv`". This command will create a new directory named '`myenv`' in the current directory, containing a complete Python environment with its own version of Python and its own package library. To activate, navigate to the created environment '`myenv`' and type "`source bin/activate`". In Windows, the activation script is in the "`Scripts`" subdirectory within the virtual environment directory and runs the command "`myenv\Scripts\activate`". Once the virtual environment is activated, we can install packages using pip and run Python scripts.

To switch between environments, we need to deactivate the current virtual

environment using the command “**deactivate**” and then activate the other virtual environment, as shown in the following *Figure 2.16*:

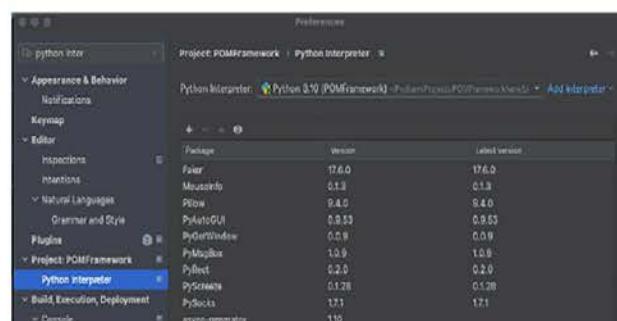
32 ■ Selenium and Appium with Python

```
(venv) yogashivamathivanan@Yogashivas-MacBook-Pro POMFramework % python -m venv myenv
(venv) yogashivamathivanan@Yogashivas-MacBook-Pro POMFramework % pip
/Users/yogashivamathivanan/PycharmProjects/POMFramework
(venv) yogashivamathivanan@Yogashivas-MacBook-Pro POMFramework % ls
Configurations          Reports           TestData           main.py
Logs                   Screenshots        Tests             myenv
Pages                  Screenshotstest_invalid_login.png Utilities
(venv) yogashivamathivanan@Yogashivas-MacBook-Pro cd myenv
(venv) yogashivamathivanan@Yogashivas-MacBook-Pro myenv % pip
/Users/yogashivamathivanan/PycharmProjects/POMFramework/myenv
(venv) yogashivamathivanan@Yogashivas-MacBook-Pro myenv % cd bin
(venv) yogashivamathivanan@Yogashivas-MacBook-Pro bin % ls
activate.ps1  activate  activate.csh  activate.fish  pip      pip3      pip3.10    python   python3  python3.10
(venv) yogashivamathivanan@Yogashivas-MacBook-Pro bin % cd ..
(venv) yogashivamathivanan@Yogashivas-MacBook-Pro myenv % source bin/activate
(myenv) yogashivamathivanan@Yogashivas-MacBook-Pro myenv % pip install selenium

(myenv) yogashivamathivanan@Yogashivas-MacBook-Pro myenv % deactivate
yogashivamathivanan@Yogashivas-MacBook-Pro myenv % ls
bin          include       lib          pyvenv.cfg
yogashivamathivanan@Yogashivas-MacBook-Pro myenv % cd ..
yogashivamathivanan@Yogashivas-MacBook-Pro POMFramework % ls
Configurations          Screenshots        Utilities         venv
Logs                   Screenshots        main.py
Pages                  TestData          myenv
Reports                Tests            myenv2
(venv) yogashivamathivanan@Yogashivas-MacBook-Pro POMFramework % cd venv
yogashivamathivanan@Yogashivas-MacBook-Pro venv % source bin/activate
(venv) yogashivamathivanan@Yogashivas-MacBook-Pro venv %
```

Figure 2.16: Create, Activate, and Deactivate the virtual environment

The virtual environment can also be created in PyCharm by navigating to Python Interpreter and adding a new Interpreter as in the following *Figures 2.17* and *2.18*:



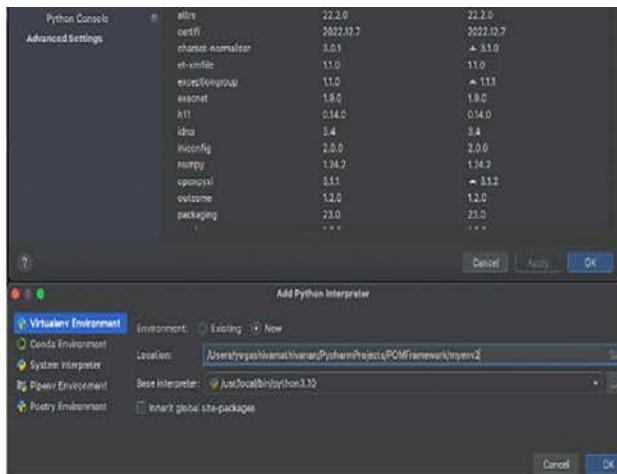


Figure 2.17: PyCharm New virtual environment -I

Refer to the following Figure 2.18:

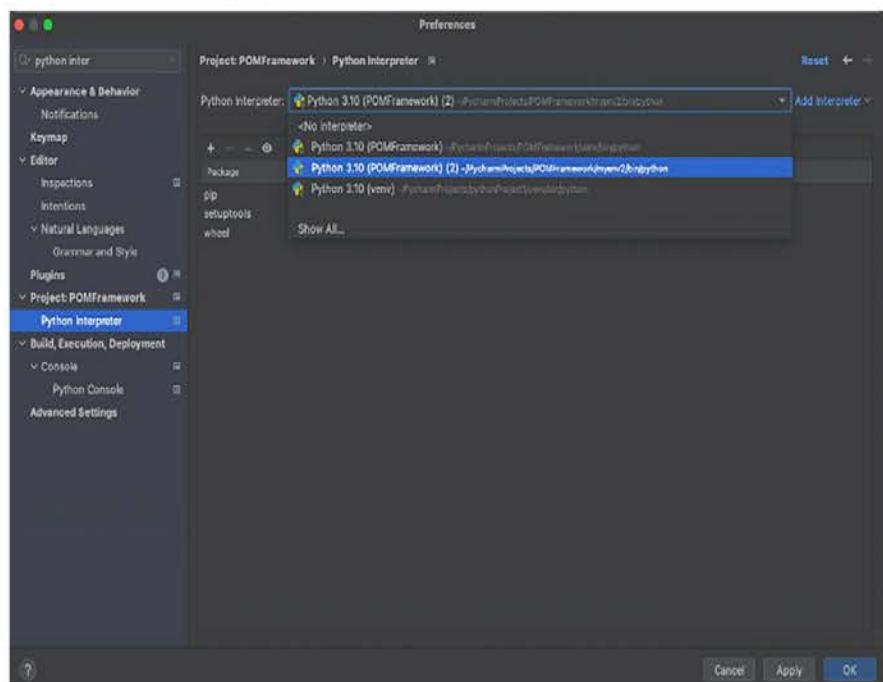


Figure 2.18: PyCharm New virtual environment -II

Requirement.txt

In Python, a '**requirements.txt**' file is a simple text file that lists all the packages and their specific versions required for the project to run correctly. The '**requirements.txt**' file is used by tools such as pip, to automatically install the correct version of each package when setting up a new environment. To generate the '**requirements.txt**' file, activate the virtual environment that we need to create the file for. Then

use pip command “`pip freeze > requirement.txt`” as shown in the following Figure 2.19. By specifying all the required packages and their versions in a single file, we can easily recreate the same Python environment on a different machine or for different users. This is useful when sharing the project or when deploying the project in different environments. Additionally, we can keep track of dependencies over time. Please refer to the following figure:

The screenshot shows the PyCharm interface with a project named 'POMFramework'. The 'requirements.txt' file is selected in the file tree. The terminal at the bottom shows the command: `venv) yogashivamathivanan@Yogashivas-MacBook-Pro venv % pip freeze > requirements.txt`.

```
1  async-generator==1.18
2  attrs==22.2.0
3  certifi==2022.12.7
4  charset-normalizer==3.0.1
5  et-xmlfile==1.1.0
6  exceptiongroup==1.1.0
7  execnet==1.9.0
8  Faker==17.6.0
9  h11==0.14.0
10 idna==3.4
11 iniconfig==2.0.0
12 MouseInfo==0.1.3
13 numpy==1.24.2
14 openpyxl==3.1.1
15 outcome==1.2.0
16 packaging==23.8
17 pandas==1.5.3
18 Pillow==9.4.0
19 pluggy==1.0.0
20 PyAutoGUI==0.9.53
21 PyGetWindow==0.0.9
22 PyMsgBox==1.0.9
23 pyobjc==9.0.1
24 pyobjc-core==9.0.1
25 pyobjc-framework-Accessibility==9.0.1
```

Figure 2.19: Creating Requirement.txt file

Core concepts of Python

There are various core concepts of Python and we will explore them in this section.

Variables and Data Types

Variables are used to store the data, and data can be of various types such as integers, complex, strings, and so on. In Python, we do not have to declare the type of the variable. This is because the Interpreter assigns the data type to the variable during execution.

The most used Python data types are Numbers, Strings, Boolean, and Lists. Numbers can be classified as Integer, Float, and Complex numbers. In Python, let us look at some examples to understand concepts such as variables, data types, assignments, typecast, and so on.

Let us refer to the following example. A variable **a** is created and assigned value 20. Here, the variable **a** is an integer and this variable is then assigned to value John. Here, type of the same variable is changed to String. To find the type of a variable, pass the variable as an argument to the function **type**.

CODE	OUTPUT
1. a = 10	1. 10
2. print(a)	2. <class 'int'>
3. print(type(a))	3. John
4. a = 'John'	4. <class 'str'>
5. print(a)	
6. print(type(a))	

The variable data type can be typecast between other data types. In the following example, value of a is "10" of type string and b is 10.0 of type float.

CODE	OUTPUT
1. a = str(10)	1. 10 10.0
2. b = float(10)	2. <class 'str'> <class 'float'>
3. print(a, b)	
4. print(type(a), type(b))	

The variable can have multiple variable assignments in a single statement. In the following example, variables, a, b and c are assigned to different data types in

sequential order, in a single line. We can have a variable placeholder in the print statement with curly braces and format to assign value or data.

CODE

```
1. a, b, c = 5, 6.4, "String"  
2. print (a, b, c)  
3. print("{} {}".format("Value is", c))  
4. print(type(a), type(b), type(a))
```

OUTPUT

```
1. 5 6.4 String  
2. Value is String  
3. <class 'int'> <class 'float'> <class 'str'>
```

The multiline string can be written within triple single quotes as in the following example.

CODE

```
1. a = '''This is  
2. the Multi-Line  
3. String assigned to  
4. Single variable'''  
5. print(a)  
6. print(type(a))
```

OUTPUT

```
1. This is  
2. the Multi-Line  
3. String assigned to  
4. Single variable  
5. <class 'str'>
```

Boolean Data type has two values: True or False. The Boolean data type is very significant in conditional programming. The variable is of type bool and the condition returns True or False, based on the statement. Refer to the following example.

CODE

```
1. a = True  
2. b = False  
3. print(a, type(a))  
4. print(b, type(b))  
5. print(4 > 6)
```

OUTPUT

```
1. True <class 'bool'>  
2. False <class 'bool'>  
3. False
```

Types of operators in Python

Operators in Python are symbols used to perform operations on the data and variables.

Arithmetic operator

To perform operations such as addition (+), subtraction (-), multiplication (*), division (/), and so on, the arithmetic operator is used. It also does operations such as modulus (%) by returning the remainder of the division, exponent (**), and floor division (//) by returning the round off number from division. However, if one of the numbers is negative, it returns round away from zero. Refer to the following examples.

CODE	OUTPUT
1. a = 15	1. 17
2. b = 2	2. 13
3. print(a+b)	3. 30
4. print(a-b)	4. 7.5
5. print(a*b)	5. 1
6. print(a/b)	6. 225
7. print(a%b)	7. 7
8. print(a**b)	8. -4
9. print(a//b)	
10. print(-11//3)	

Comparison operator

The Comparison operator is used to compare two values such as equal (==), not equal (!=), greater than (>), Lesser than (<), greater than equal to (>=), and lesser than equal to (<=). Refer to the following examples.

CODE	OUTPUT
1. print(5 == 5)	1. True

2. print(5 != 5)	2. False
3. print(5 > 1)	3. True
4. print(5 < 10)	4. True
5. print(5 >= 5)	5. True
6. print(5 <= 5)	6. True

Assignment operator

The Assignment Operator is used to assign values to the variable. Add & assign ($+=$) adds the variable to the value and assigns the result to the variable. Other operations include like assign ($=$), subtract & assign ($-=$), multiple & assign ($*=$), modulus & assign ($%=$), exponent & assign ($**=$), floor division and assign ($//=$) and so on. Let us look at some examples of this.

CODE	OUTPUT
1. a = 15	1. 13 2 27
2. x, y, z=10, 5, 3	
3. x += 3	
4. y %= 3	
5. z **= 3	
6. print(x, y, z)	

Logical, identity, and membership operator

These operators are used for conditional statements (and, or, not), for comparing objects (is, is not), and verifying if sequence is present in an object (in, not in) respectively. We will see them in detail in further sections.

Lists in Python

Lists in Python is the most versatile data type, used to store the collection of data in a single variable. Data in the list are stored in square brackets separated by commas. A list can contain different data types and allows duplicate values, and list items are

changeable and sequenced. To access the value from the list, use the square bracket with the index or position of the value with starting index of 0. A negative index means the count starts from the end with the last index being -1. There are some functions used to manipulate the list such as append and insert, that are used to add values, remove, and function `len` to find the length of the list. Following are some examples of working with the list. Additionally when we use `myList[::-1]`, it means that we are slicing the entire list with a step count of 1, which means it will print every element of the list without skipping any, equivalent to printing the original list. If we want to slice the list and print every alternative value of the list, we can specify a step count of 2 in the slice notation '`myList[::-2]`'.

CODE

```
1. myList = ["One", "Two", "Three"]
2. print(myList)
3. print(type(myList))
4. myList = [1, 2, "Mango", 8.4, "Apple"]
5. print(myList)
6. print(myList[2])
```

```
7. print(myList[-1])
8. print(myList[1:3])
9. print(myList[-3:-1])
10. myList.insert(15, "Orange")
11. myList.append(19.5)
12. print(myList)
13. myList[1] = "Grapes"
14. print(myList)
15. myList.remove(8.4)
16. print(myList)
17. print(len(myList))
18. print(myList[::-2])
```

OUTPUT

```
1. ['One', 'Two', 'Three']
2. <class 'list'>
```

3. [1, 2, 'Mango', 8.4, 'Apple']
4. Mango
5. Apple
6. [2, 'Mango']
7. ['Mango', 8.4]
8. [1, 2, 'Mango', 8.4, 'Apple', 'Orange', 19.5]
9. [1, 'Grapes', 'Mango', 8.4, 'Apple', 'Orange', 19.5]
10. [1, 'Grapes', 'Mango', 'Apple', 'Orange', 19.5]
11. 6
12. [1, 'Mango', 'Orange']

Note line 10 in the preceding code, where we try to insert Orange in the 15th index but the output at line 8 shows Orange is automatically added to the 5th index. Because of the sequential nature of the list, data will be added to the next available index.

Tuples

Tuples in Python are like Lists. They are used to store the collection of data in a single variable. However, where Lists are mutable, Tuples are immutable, meaning, tuples cannot be changed after the declaration because they are protected. Data in the tuple are stored in round brackets separated by commas. Unlike a list, a tuple object does not support item assignment. Let us look at some examples.

CODE	OUTPUT
1. val = (6, 7, 88.67, "Tuple")	1. Tuple
2. print(val[3])	

Sets

In Python, a set is an unordered collection of unique elements. It is represented by curly braces '{}' or the built-in 'set()' function. Sets are like lists and tuples in that they can contain multiple elements, but they are different in that they cannot contain duplicate elements. Thus, it can be used to remove duplicate values in lists or tuples

by converting them to a set. The main advantage of a set is its ability to perform set operations such as union, intersection, difference, and symmetric difference. These operations are implemented using mathematical set theory principles.

Sets in Python are mutable, as we can add or remove elements for a set after it has been created using the `add()` and `remove()` methods, respectively.

In the following code, we are creating a set, creating a set from integers, a tuple of strings, and a string. We can combine sets using union operation, and find common elements using the `intersection()` function, and difference using the `difference()` function which returns elements in the first set but not in the second set. Lastly, `symmetric_difference()` returns elements that are in the first set or second set but not in both.

CODE

```
1. mySet = {1, 2, 3, 4, 5}  
2. mySet2 = {"apple", "banana", "cherry"}  
3.  
4. print(type(mySet))  
5. print(type(mySet2))  
6.
```

```
7. # Creating a set from a list of integers  
8. myList = [1, 2, 3, 4, 5, 5]  
9. mySet = set(myList)  
10. print(mySet)  
11.  
12. # Creating a set from a tuple of strings  
13. myTuple = ("apple", "banana", "cherry", "apple")  
14. mySet2 = set(myTuple)  
15. print(mySet2)  
16.  
17. # Creating a set from a string  
18. myString = "hello"  
19. mySet3 = set(myString)  
20. print(mySet3)
```

```

20. print(myset)
21.
22.
23.
24. # Union operation
25. set1 = {1, 2, 3}
26. set2 = {3, 4, 5}
27. union_set = set1.union(set2)
28. print(union_set)
29.
30. # Intersection operation
31. set3 = {3, 4, 5}
32. intersection_set = set2.intersection(set3)
33. print(intersection_set)
34.
35. # Difference operation
36. set4 = {1, 2, 3, 4, 5}

```

42 ■ Selenium and Appium with Python

```

37. set5 = {4, 5, 6, 7, 8}
38. difference_set = set4.difference(set5)
39. print(difference_set)
40.
41. # Symmetric difference operation
42. symmetric_difference_set = set4.symmetric_difference(set5)
43. print(symmetric_difference_set)

```

OUTPUT

```

1. <class 'set'>
2. <class 'set'>
3. {1, 2, 3, 4, 5}
4. {'banana', 'cherry', 'apple'}

```

5. {'l', 'e', 'o', 'h'}
6. {1, 2, 3, 4, 5}
7. {3, 4, 5}
8. {1, 2, 3}
9. {1, 2, 3, 6, 7, 8}

Dictionaries

Dictionaries are the type of data structure that stores the data in the form of Key-Value pair. A dictionary is changeable; we can add, remove or update items after declaring it. If it is ordered, the order of the items cannot change. Neither does it allow duplicates; two values cannot have the same keys. The following examples shows how to create a dictionary, update it, add items, and delete it.

CODE	OUTPUT
1. employeeDetails = {1: "Employee",	<class 'dict'>
2. "EmployeeID": 12345,	12345
3. "first_name": "John",	{'EmployeeID': 12345,
4. "age": 32}	'first_name': 'John', 'age': 30, 'salary':
5. print(type(employeeDetails))	6050.99}
6. print(employeeDetails['EmployeeID'])	

7. employeeDetails["age"] = 30
8. employeeDetails["salary"] = 6050.99
9. del employeeDetails[1]
10. print(employeeDetails)

Conditional statements in Python

There are various conditional statements in Python, and we will explore them in this section.

IF-ELSE Condition

"**IF**", "**ELIF**" and "**ELSE**" keywords are used to execute a statement if the condition is true, if previous conditions are not true, and if no condition is true respectively. The Conditions can also be combined using "**And**" and "**Or**" keywords and we can

have nested conditions as well, meaning “if” condition within “if” condition. Let us look at some examples.

Indentation

The scope of the code in Python is defined using a white space at the beginning of the line called indentation. In the following example, in the line after the “if” condition, if the statement is written without a space in the beginning, it will throw an error: “**IndentationError: expected an indented block after 'if' statement**”.

CODE

```
1. a = 50
2. b = 100
3. c = 200
4. if a > b:
5.     print("a is greater than b")
6. elif a > c:
7.     print("a is greater than c")
8. elif a < c and a > b:
9.     print("a is greater than b and less than c")
10. elif c < b or a > c:
```

```
11.     print("a is greater than c or c less than b")
12. else:
13.     print("Above all conditions are false")
14.
15. if a >= 50:
16.     print("a is above or equal to 50")
17.     if c >= b and c >= a:
18.         print("c is greater than a and b")
19.     else:
20.         print("c is greater than a and b")
```

OUTPUT

1. Above all conditions are false
2. a is above or equal to 50
3. c is greater than a and b

Loops in Python

Loops in a programming language are written to run certain statements iteratively until the defined range or the condition is true. Major loops provided in Python are the **While** loop and **For** loop.

WHILE Loop

The while loop is used to iterate over a block of code multiple times until the set condition is true. In the **while** loop, if the condition is true then the loop executes the code. After one iteration, the condition is checked again before executing the code. If the condition fails, then the execution exits the loop. Here, indentation is important.

The **BREAK** Statement is used to stop the iteration and exit the loop even if the condition is true. In the second example in the following code, the code exits the **while** loop when the value of the variable *a* was less than 3.

The **CONTINUE** statement is used to stop the current iteration and continue to the next one. In the third example in the following code, the code skipped the iteration when the value is equal to 3. The **While** loop can also be used with the **else** condition, which will be executed once when the **while** loop condition fails, as in the following fourth example. Now, let us look at the syntax with examples.

CODE	OUTPUT
1. flag = True	1. 20
2. sum = 0	2. 40
3. while flag:	3. 60
4. sum += 20	4. 60
5. print(sum)	
6. if sum >= 50:	
7. flag = False	
8. print(sum)	

CODE	OUTPUT
------	--------

1. a = 5	1. 4
2. while a > 0:	2. 3
3. a -= 1	
4. if a < 3:	
5. break	
6. print(a)	

CODE	OUTPUT
1. a = 5	1. 4
2. while a > 1:	2. 2
3. a -= 1	3. 1
4. if a == 3:	
5. continue	
6. print(a)	

CODE	OUTPUT
1. a = 1	1. 1
2. while a < 3:	2. 2
3. print(a)	3. a is not greater than or equal to 3
4. a += 1	
5. else:	
6. print("a is not greater than or equal to 3")	

FOR Loop

FOR loop in Python is used to iterate over a sequence (lists, tuples, dictionary, string, and so on) or iterable objects. **FOR** loop iterates till the end of the sequence. The keywords **BREAK**, and **CONTINUE** can be used to exit or skip the current iteration respectively.

CODE	OUTPUT
1. myList = {"apple", "mango", "grapes", "orange"}	1. apple
2. for i in myList:	2. mango
3. print(i)	3. grapes
4. }	4. orange

```

5.     print(i)                      5. grapes
6.                                         4. orange
5. firstString = "car"                5. c
6. for i in firstString:
7.     print(i)                      6. a
8.                                         7. r
9. for i in range(3):
10.    print(i)                     8. 0
11.                                         9. 1
12.                                         10. 2

```

FOR ELSE loop

The **FOR ELSE** loop is used to execute a block of code when a loop completes normally, that is, without being interrupted by a 'break' statement.

In the following example, the **FOR** statement starts a loop that iterates over each item in the iterable object '**myList**'. On each iteration, the value of the current item is assigned to the variable 'item'. The indented block of code under the 'FOR statement is executed on each iteration of the loop. If the loop completes normally then the indented block of code below the 'else statement is executed.

CODE

```

1. myList = [1, 2, 3, 4, 5]
2. searchValue = 6
3.
4. for item in myList:
5.     if item == searchValue:
6.         print("Item found!")

```

```

7.         break
8.     else:
9.         print("Item not found.")

```

OUTPUT

1. Item not found.

Functions in Python

Functions in Python is a block of code used to perform a single, related action. The code can perform the code action and return the result, if required, every time the function is called. Thus, it helps in code reusability and modularity. We have used the `print()` function in all the examples to print items to the console, which is an inbuilt function provided by Python. Likewise, we can create a user-defined function to perform certain tasks.

A Function in Python begins with keyword “`def`”, and then resumes with function name with enclosed parenthesis, to pass the arguments to the function followed by colon “`:`”. Then in the next line, code can be written with an indentation.

Multiple arguments passed to a function are separated by “`,`”. The function needs to be called with all the arguments, as expected by the function, but in case the number of arguments is unknown, then add `*` before the argument. This way, the function can accept a tuple of arguments. The created function needs to be called from outside for execution, which uses the name of the function followed by a parenthesis.

Global variables and local variables

The variables declared within the function are local variables that can only be accessed within the function body, whereas the variables declared globally are accessible outside the function as well. In the following example, the `global_var` variable declared globally can be accessed in `access_global_var()`: function and updated to 5000, but the scope of change to global variable is limited to the function itself. As in line 34 in the following example, when we print `global_var`, the value is still 100.

CODE

```
1. global_var = 100
2.
3.
4. def addition_of_two_numbers(first_number, second_number):
5.     result = first_number + second_number
6.     return result
7.
8.
```

```

9. def calculate_average_and_grade(first_sub=0, second_sub=0, third_
sub=0):
10.     grade = None
11.     average = 0
12.     if 0 <= first_sub <= 100 and 0 <= second_sub <= 100 and 0 <=
first_sub <= 100:
13.         average = round((first_sub + second_sub + third_sub) / 3, 2)
14.         if average >= 80:
15.             grade = "A"
16.         elif average >= 60:
17.             grade = "B"
18.         else:
19.             grade = "C"
20.     else:
21.         average = 0
22.         grade = "C"
23.     return [average, grade]
24.
25.
26. def access_global_var():
27.     global_var = 5000
28.     print(global_var)
29.
30.
31. print(addition_of_two_numbers(89, 76))

```

```

32. print(calculate_average_and_grade(second_sub=97, third_sub=95))
33. access_global_var()
34. print(global_var)

```

When the first function **addition_of_two_numbers** is called, the two arguments are not named. This makes the argument ordered, meaning the first argument is **first_number** and the second argument is **second_number**.

When we call the second function `_average_and_grade`, the arguments default to 0, and arguments are passed in an unordered structure. This is called a keyword argument, where the order is not necessary. If the argument is not provided, then the function will take the default value.

OUTPUT

1. 165
2. [64.0, 'B']
3. 5000
4. 100

Modules, packages, exception handling in Python

In this section, we will be exploring modules, packages and exception handling in Python.

Modules

In Python, Modules are code library files with `.py` extensions, containing a set of functions. The module can be imported to utilize the containing block of code in the application. The module contains the implementation of classes, variables, and Functions.

The module increases reusability, simplifies the code, and defines scope within Python. Python already has standard libraries with functions, so when imported, it saves a lot of time. These Python libraries can be utilized within the code using the `import` keyword and to get specific functions from a module, we use the `from` keyword.

In the following example, the module `ModuleImport` is imported in another Python module using keyword `from ModuleImport import ModuleImport`. This keyword imports variables, functions from `ModuleImport`.

CODE

1. class ModuleImport:

```

2.
3.     module_variable = "variable"
4.
5.     def message_from_my_module(self, message):
6.         print("This is the Message: " + message)

```

CODE

```

1. from ModuleImport import ModuleImport
2.
3. ModuleImport().message_from_my_module("Message from imported
Module")
4. print(ModuleImport().module_variable)

```

For an example of an inbuilt module, let us consider commonly used module “math”.

CODE	OUTPUT
1. import math	1. 3.0
2.	2. 20736.0
3. print(math.sqrt(9))	
4. print(math.pow(12, 4))	

Packages

A Python package is a directory used to store the modules in one package, that might contain sub-packages, modules, sub-modules, and so on. For a Python interpreter to identify a directory as a package, the directory contains the file `__init__.py`, which holds the import statement of all the modules, sub-modules, and sub-packages. Refer to *Figure 2.20*:

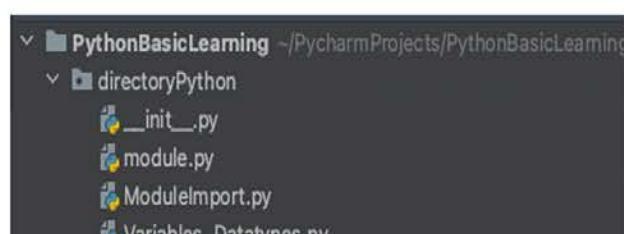


Figure 2.20: Python Directory

Errors and exceptions in Python

The Errors are defined as irregularities or mistakes within the code, like format errors, and calling undeclared variable error, which will stop the execution of the program. On the other hand, exceptions occur when the program is syntactically correct, but an error occurs during execution. Thus, two types of errors occur in Python: Syntax errors and Logical errors (called exceptions).

Syntax Error

CODE	OUTPUT
1. score = 1000	1. while score >= 500
2.	2. ^
3. while score >= 500	3. SyntaxError: expected ':'
4. print(score)	
5. score -= 100	

Exceptions

CODE	OUTPUT
1. score = 1000	1. print(score/0)
2.	2. ZeroDivisionError: division by zero
3. while score >= 500	
4. print(score/0)	
5. score -= 100	

Some common exceptions are as follows:

- **IndentationError:** Indentation is not specified properly.
- **ZeroDivisionError:** Raised when division by zero.

- **ArithmeticError**: Base class for all errors that occur for numeric calculation.
- **Exception**: Base class for all exceptions.
- **ValueError**: When function gets correct argument type, but incorrect value.
- **IndexError**: Access item out of range.
- **AttributeError**: When attribute assignment fails.
- **KeyError**: When the key in the dictionary is not found.
- **MemoryError**: When the program runs out of memory.
- **ImportError**: When the imported module is not found.

Exception Handling

We can handle these exceptions, so that our program does not stop the execution. Python provides Try-**Except-Else-Finally** Block to handle exceptions. Refer to the following code:

CODE

```

1. class Exception_Handling:
2.
3.     def exceptions_handling_python(self, first_number, last_number):
4.         try:
5.             division_number = first_number / last_number
6.         except ZeroDivisionError:
7.             print('Division by 0 is not possible')
8.         else:
9.             print(division_number + "is the result")
10.        finally:
11.            print("Finally Block: will be executed always")
12.
13.    def raise_exception_python(self, number):
14.        try:

```

```

15.            if (number > 100):
16.                raise ArithmeticError("Number in the Function cannot
be greater than 100")

```

```

17.         except ArithmeticError:
18.             print(ArithmeticError)
19.         else:
20.             print(str(number) + " is less than or equal to 100")
21.     finally:
22.         print("Finally Block: will be executed always")
23.
24.
25.
26.
27. Exception_Handling().exceptions_handling_python(5, 0)
28. Exception_Handling().raise_exception_python(150)

```

In the preceding example, we call the function `exceptions_handling_python` with argument “`last_number`” as zero. Instead of an exception in try block, the exception is caught by except block. If no exception is raised (when “`last_number`” is non-zero) the control goes to the `else` block without executing the code in `except` block. Code within `Finally` block will be executed irrespective of whether an exception is raised or not.

We can also raise an exception using the `raise` keyword as in the function `raise_exception_python`. The function is called with argument greater than 100 and we are raising `ArithmeticError` exception and handling them in except block.

OUTPUT

1. Division by 0 is not possible
2. Finally Block: will be executed always
3. <class 'ArithmeticError'>
4. Finally Block: will be executed always

CLASSES AND OBJECTS IN PYTHON

Python is an object-oriented programming language. Everything in Python is an object, and a class in Python is like an object constructor. Classes have attributes that include class variables and instance variables, as well as methods that modify these attributes. The class is created using the keyword `class`.

The `__init__` method is a class constructor; this method is invoked when an object for the class is created and used to initialize all the required variables in the class. The `Self Keyword` represents an instance of the class and is used to allow access to the attributes and methods of the class.

CODE

```
1. class Vehicle:  
2.     def __init__(self, company, model, door):  
3.         self.company = company  
4.         self.model = model  
5.         self.door = door  
6.         self.range = 350  
7.         self.remainingRange = 0  
8.  
9.     def complete_charge_car(self):  
10.        self.remainingRange = self.range  
11.        print("Car is fully Charged")  
12.  
13.    def drive(self, miles):  
14.        print(f'Driving {self.model}')  
15.        self.remainingRange -= miles  
16.  
17.    def plug_in_charge(self, after_charge):  
18.        if after_charge <= self.range:  
19.            self.remainingRange = after_charge  
20.        else:  
21.            print('Charge Done - Unplug')
```

```
23.  
24. ev_object = Vehicle('Tesla', 'model 3', 4)  
25. print(ev_object.model)  
26. ev_object.complete_charge_car()  
27. print(ev_object.remainingRange)  
28. ev_object.drive(150)  
29. print(ev_object.remainingRange)  
30. ev_object.plug_in_charge(300)  
31. print(ev_object.remainingRange)
```

Inheritance in Python

Inheritance is a powerful feature of **Object-Oriented Programming (OOP)** that allows the creation of new classes inheriting features of the existing class, enabling code reuse, extension. Customization also enables extending and customizing the behavior of existing classes.

In Python, we can create a new class that inherits from an existing class by specifying the existing class in parentheses after the new class name. In the following code, the 'Animal' class is the base class, and 'Dog' and 'Cat' classes are derived. The derived classes inherit all the attributes and methods of the base class and can add their own unique attributes and methods as well. To call a method from the base class within a derived class, we can use '`super()`' function. In the following example '`super()`' function in '`__init__()`' method of 'Dog' class is used to call '`__init__()`' method of the 'Animal' class to set the 'name' attribute.

CODE

```
1. class Animal:  
2.     def __init__(self, name):  
3.         self.name = name  
4.  
5.     def speak(self):  
6.         print(self.name + " is speaking.")  
7.  
8.
```

```
9. class Dog(Animal):
10.     def __init__(self, name):
11.         super().__init__(name)
12.
13.     def speak(self):
14.         print(self.name + " is speaking Bow.")
15.
16.
17. class Cat(Animal):
18.     def __init__(self, name):
19.         super().__init__(name)
20.
21.     def speak(self):
22.         print(self.name + " is speaking Meow.")
23.
24.
25. my_animal = Animal('Peso')
26. print(my_animal.name + " is an animal!")
27. my_animal.speak()
28.
29. my_dog = Dog('Willie')
30. print(my_dog.name + " is a dog.")
31. my_dog.speak()
```

OUTPUT

1. Peso is an animal!
2. Peso is speaking.
3. Willie is a dog.
4. Willie is speaking Bow.

Conclusion

Python is the most popular and widely used general-purpose, high-level object-oriented programming language. Application of Python is everywhere in modern technology. With an understanding of the core concepts of Python, we are ready to step into software automation using Selenium Leveraging Python programming language. We will go over the setup and the core concepts of Selenium in the following chapter.

Key facts

- Python is an object-oriented programming language.
- Python interpreter is a virtual machine that is like a real computer, which compiles and executes the source code.
- Variable in Python is used to hold data type numbers, string, and Boolean values. To assign values to a variable, we do not need to specifically declare the variable type during runtime.
- List, Tuple, Set, and Dictionary are the built-in data types used to store multiple values in a single variable.
- Operators in Python are used to do operations. This includes Arithmetic, Assignment, Comparison, Logical, Identity and membership operators.
- Conditional statements and loops help to decide the flow and iteration of execution respectively.
- Functions are the block of code used to perform one action.
- Class is a structure from which objects are created.
- Two types of errors and exceptions are syntax errors and logical errors (called exceptions).

Questions

1. What is Local and Global Variable?
2. What is an environment variable? How do you create, activate and deactivate a virtual environment?
3. What is a `requirement.txt` file and what is its significance?
4. What is the difference between a List and Tuple?
5. What are Sets in Python? How do you use sets to remove duplicates from List or a Tuple?

6. What is the difference between a List and a Dictionary?
7. What is the difference between the Break and Continue keywords?
8. What is a class method and static method in Python?
9. What is inheritance in Python? Explain.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



CHAPTER 3

Selenium for Web Automation

Introduction

Selenium is an open-source web application automation tool, that supports all major browsers. Selenium automates human interaction with web elements in an application in the browser. Web elements can be a text box, button, dropdown selection, radio button, browser tabs, alert pop-up, and so on. The interactable action in an application can vary from clicking, selecting, dragging, and dropping, or typing in the web elements. Selenium is the optimum solution for automating functional and regression test cases. Test scripts can be written in different programming languages such as Python, java, c#, PHP, Ruby, Perl and. JavaScript.

Selenium with Python has more advantages compared to other programming languages. Python is easier with less code, packages and indentations, and faster program run time, and Python API enables connecting to browsers through Selenium.

Selenium is a widely used tool and recently a new version of Selenium was released, that is, Selenium 4, with advanced features addressing key issues. In this chapter, we will work with both Selenium 3 and Selenium 4 to help understand the differences.

Structure

In this chapter, we will discuss the following topics:

- What is Selenium?
- Selenium Webdriver Architecture
- Selenium WebDriver installation and setup
- Basic Code: Browser Invoke and Windows setup
- First Automation: Login Scenario
- Webdriver Manager
- Webdriver Manager vs Driver Path

Objectives

This chapter will explain the benefits of using Selenium for web automation with Python and will go over the basic structure of Selenium, Selenium installation, and set up with browser invoking. We will apply Python learning in automation testing and learn how to leverage the Python Selenium web driver client library, to access Selenium-provided methods to automate the test cases in the browser.

Since we are using Python as a programming language to automate, upgrading to the latest Selenium version, that is, Selenium 4, is a painless process. In this chapter, we will detail the key differences between Selenium 3 and Selenium 4 versions. This will help to work in both Selenium 3 and Selenium 4, as per the project demand and upgrade if necessary.

What is Selenium?

Automation testing aims to limit the time spent on manual efforts and focus on more accurate and reliable test results. Selenium is one such powerful open-source automation tool that fulfils all the automation requirements and expectations. Selenium is a suite of tools.

Selenium was developed in 2004 by Jason Huggins, who initially named it JavaScript TestRunner. He later realized its potential, and renamed it to Selenium Core. Next is Selenium **Remote Control (RC)** developed by Paul Hammant, to solve domain-related issues while testing web applications. It was further upgraded to develop Selenium Grid for parallel testing purposes followed by Selenium IDE with record and playback function. In 2008, Selenium merged webdriver with RC and called

Selenium 2, which upgraded to Selenium 3 and later to Selenium 4 on October 13, 2021.

Selenium is a suite of software tools to automate UI-based test cases in the browser. Let us look at the broad classification of the Selenium suite in the diagram shown in *Figure 3.1*, before deep diving into different types of testing. Please refer to the following figure:

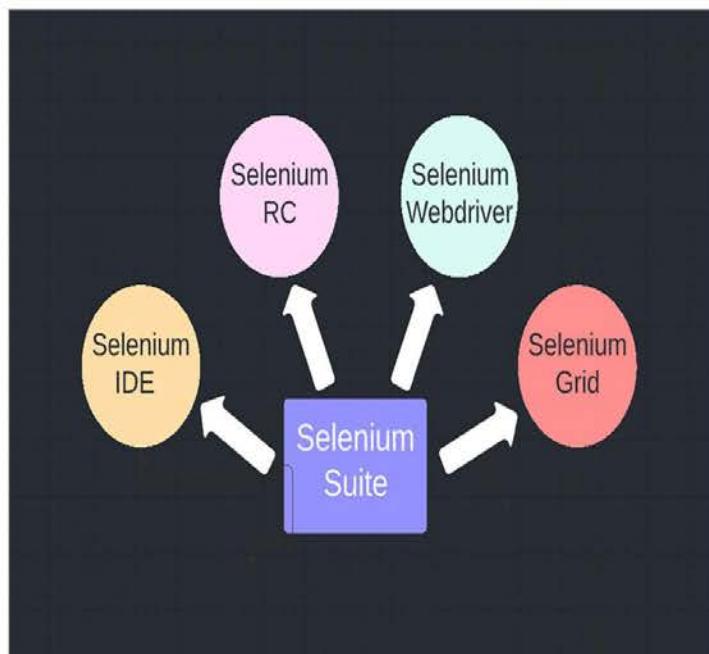


Figure 3.1: Different tools in selenium Suite

The classification are as follows:

- **Selenium Integrated Environment (IDE):** It is the simplest and most easy framework in the Selenium suite and requires no programming experience. It is a Firefox plugin that can be installed and used to record the interaction action with a click of a button and can be played back. IDE is not supported in other browsers and cannot be used for complex test cases.
- **Selenium Remote Control (RC):** Also known as Selenium1, it was the flagship of the testing framework, and runs and launches all the browsers in the server one at a time. It was the first tool to allow the use of multiple programming languages. Selenium RC has faster execution than IDE.
- **Selenium Webdriver:** The breakthrough that allows direct communication with the browser. It is an advanced version of Selenium RC that implements the modern automation approach, supports multiple programming

interfaces, and various OS, and also supports cross-browser testing. We will be using Selenium Webdriver in this chapter for automation.

- **Selenium Grid:** It allows us to run our test scripts in multiple browsers and operating systems on remote machines, by routing commands to remote browser instances in parallel.

Advantages of Selenium

The advantages of Selenium are as follows:

- **Open-Source:** Selenium is an open-source automation tool, and so Selenium is publicly accessible for free with no cost for either the tool or the support. The Selenium community is continuously growing and evolving, helping software engineers in test automation efforts.
- **Programming Languages and Framework Support:** Language and Framework support, since Selenium supports all major languages including Java, Python, Ruby, JavaScript, C#, and Perl for automation. This allows writing the automation script in any of these programming languages and Selenium converts to Selenium-compatible code in real-time. Moreover, both **Behavior-Driven Development (BDD)** and **Test-Driven Development (TDD)** frameworks are supported and can be integrated with Selenium.
- **Multi-Platform Support:** Selenium script can run on all operating systems such as Windows, Mac, Linux, Unix, and so on.
- **Multi-Browser Support:** Along with programming language and platform flexibility, selenium supports almost all majorly used browsers such as Chrome, Firefox, Opera, Safari, Edge, and Internet Explorer.
- **Add-ons/Third-Party Tool Support:** To increase the scope and efficiency of testing, Selenium supports and can be integrated with tools such as TestNg, Junit for asserting tests and report generation, Cucumber for behavior-driven development, Jenkins for continuous integration and continuous deployment, Maven & Gradle for build management tool, and so on.
- **Mobile Automation support:** With continuous updates, Selenium also supports Android and IOS application automation and web application in mobile browsers using Appium. Appium is an open-source tool with the same API as Selenium WebDriver, and one can use Selenium skills for mobile automation.
- **Easy to Implement and Reusability:** Selenium provides a user-friendly interface to automate the test scenarios effectively. Selenium test can be

executed in headless mode, that is, with the browser window minimized.

- **Parallel Test Execution:** Using the Selenium grid, the test script can be run in parallel, which helps save overall execution time. With the help of cloud grids for cross-browser testing, test scripts can be run in multiple browsers at the same time.

Selenium compared to other testing tools

Table 3.1 compares Selenium with other common tools:

	Selenium	TestComplete	Tricentis Tosca	Katalon	UFT
Release Year	2004	1999	2006	2015	1998
Open source	Yes	No	No	Yes	No
Platform Support	Win/Mac/ Linux	Win	Win	Win/ Mac	Win
Language Support	Java, Python, c#, JavaScript, Ruby, Pearl	VB, JavaScript, C++, C#, Angular, Ruby, PHP	JavaScript	Java/ Groovy	VBScript
Testing	Web, Mobile	Web, Mobile, Desktop	Web, Mobile	Web, Mobile	Web, Mobile, Desktop

Table 3.1: Comparing Selenium to other common testing tools

Selenium Webdriver Architecture

Selenium Webdriver is the most widely used component of the selenium suite. Selenium WebDriver integrated with Webdriver API provides a programming interface to support multiple programming language test scripts.

Before deep dive into Selenium Webdriver architecture, let us briefly understand how web applications work, which will help us understand the Selenium architecture better. In simple terms, when a user types the URL of any application, the browser brings the respective web page. Here, the user is the client, the browser is the connecting platform, and the web page is delivered from the server-side, as depicted in Figure 3.2:



Figure 3.2: Components of Web Application Architecture

Layers of Web application

The four layers of web applications are as follows:

- **Presentation Layer (PL):** This is the graphical user interface and communication layer for the client/user to interact with the application. This layer provides the web page with necessary data to the client-side and collects the data from the user. The layer requires and runs on a web browser.
- **Data Service Layer (DSL):** This is the back-end layer that provides an abstraction layer to hold and serve application data, and then it transmits the business logic processed data back to the presentation layer. This layer also ensures data security by separating the business and presentation logic.
- **Business Logic Layer (BLL):** The layer defines business rules and controls the functionality of the application by ensuring data exchange. The layer accepts a request from the browser, executes business logic associated with the request, and routes them to the corresponding server, and then sends the response back to the presentation layer.
- **Data Access Layer (DAL):** This layer is connected closely with BLL and has a database server that retrieves data from the corresponding server to allow direct access to the data stored in the database. This layer is used to perform data manipulation operations such as create, delete, update, and read.

Selenium 3 architecture

Selenium 3 supports **JavaScript Object Notation (JSON)** Wire Protocol to

communicate from the client to the server over **Hypertext Transfer Protocol (HTTP)**. Selenium 3 Architecture consists of five layers, which are as follows:

- **Selenium Client Library:** Selenium client Library or language bindings consists of Selenium commands in any programming language, in compliance with the **World Wide Web Consortium (W3C) Selenium protocol**. When the test case is triggered, the entire Selenium code is converted into JSON format.
- **Selenium API:** API stands for **Application Peripheral Interface**. Selenium API is a set of commands to interact with the WebDriver that allows the programs to communicate with the web application in the browser to mimic the real user actions.
- **JSON Wire Protocol:** JSON is used in communication between two components of either the same application or a different applications. The Selenium WebDriver uses JSON to communicate between the client library and browser drivers. The JSON request from the client library is converted into an HTTP request and sent to the server, then converted back to JSON before sending it to the client. This protocol enables the server to communicate with client libraries irrespective of the client library programming language.
- **Browser Drivers:** This is an intermediate component between Selenium libraries and the actual browsers. Each browser has separate drivers and is required to be invoked as the first step of Selenium code to decide which browser to run the test.
- **Browsers:** Selenium supports major browsers including Chrome, Firefox, Safari, Edge, Internet Explorer, and Opera browser.

The Selenium 3 Architecture is shown in *Figure 3.3:*





Figure 3.3: Selenium 3 Architecture

Selenium 4 architecture

Selenium 4 provides direct communication between Selenium Client and Browser Driver. The Selenium client sends a command execution request and the WebDriver language bindings is a code library that is designed to drive the user action. The browser driver executes the automation script on one of the browsers as directed by the client request and then returns a response after executing the script.

The programming language has its bindings for client and WebDriver binding. Because of the bindings, the same commands that are written for one language are also written for another.

The built-in automation support of each web browser is utilized by the WebDriver to operate the browser drivers and web browsers. The Selenium 4 architecture is shown in *Figure 3.4*:



Figure 3.4: Selenium 4 Architecture

Selenium 4 advantages

The common benefits of Selenium 4 over Selenium 3 are as follows:

- **Stability:** In Selenium 4, the webdriver APIs adopt the W3C standardization and the JSON wire protocol is removed. There is no additional encoding and decoding of the API request required, which makes Selenium 4 more stable.
- **Better management:** More control of tabs and windows in Selenium 4.
- **Improved Documentation:** The Selenium 4 official documentation is easier to navigate. Upgrading to Selenium 4 is well documented on the Selenium official page: https://www.selenium.dev/documentation/webdriver/getting_started/upgrade_to_selenium_4/.
- **Standards:** Makes the test script more reliable in all browsers. Since Selenium 4 is W3C WebDriver compatible, the request's encoding and decoding are not necessary.
- **Updated and easier locators:** Selenium 4 has improved the way of finding elements with the inclusion of relative locators such as "To left of", "Above" and so on.
- **Monitoring:** Selenium 4 has enhanced the logging and tracing process for better debugging of errors.

Selenium WebDriver installation and setup

With the understanding of Selenium architecture, let us move on to the setup and installation of Selenium, WebDriver, and browser window, and automate a simple login test case. Installation of Python and PyCharm is a prerequisite for the Selenium setup process. Since we have already discussed that in the previous chapter, the next step is to install Selenium.

Follow the given steps to install Selenium. The process is similar for Mac and Windows.

1. Make sure Python is installed and open a new project in PyCharm. Open Terminal on Mac and command prompt in Windows, and type "`python3 - version`". Open a new project in PyCharm as shown in *Figure 2.9* and *Figure 2.10* in *Chapter 2, Python Programming - Setup and Core Concepts*.
2. Navigate to the Selenium download page <https://www.selenium.dev/downloads/> and click on Python's latest stable version, as shown in *Figure 3.5*.

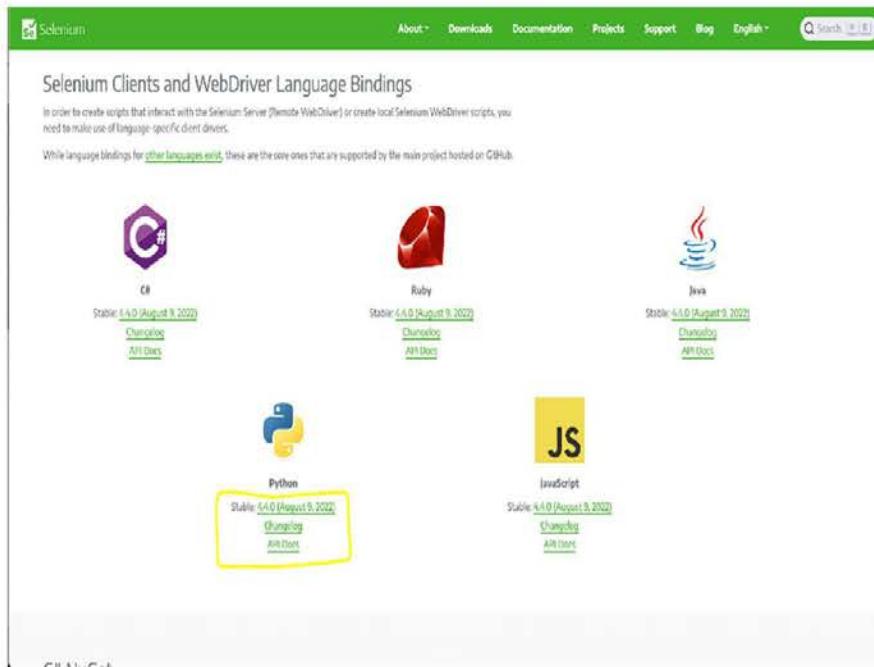


Figure 3.5: Selenium Download - Python Installation

3. The link will navigate to <https://pypi.org/project/selenium/> with the pip command, to install selenium “`pip install selenium`”, as shown in Figure 3.6:

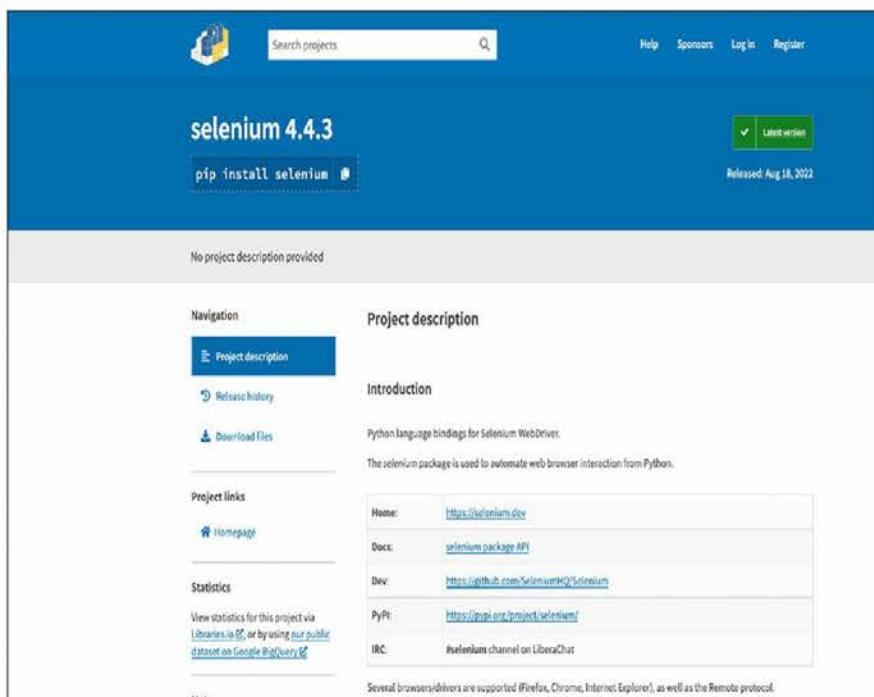
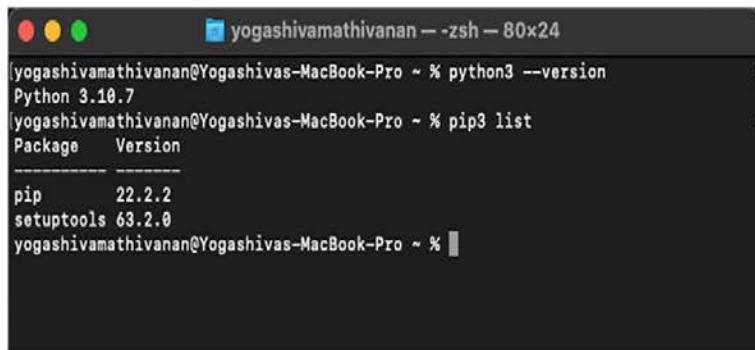


Figure 3.6: Python package installer page for selenium

- Verify that Selenium is not installed using the pip command “**pip list**” and navigating to the Python interpreter in PyCharm, as in *Figure 3.7*:



```
yogashivamathivanan@Yogashivas-MacBook-Pro ~ % python3 --version
Python 3.10.7
yogashivamathivanan@Yogashivas-MacBook-Pro ~ % pip3 list
Package      Version
-----
pip          22.2.2
setuptools   63.2.0
yogashivamathivanan@Yogashivas-MacBook-Pro ~ %
```

Figure 3.7: Verifying selenium is not downloaded using the command line

- Once the selenium is installed, the Python interpreter should have Selenium in the list, as shown in *Figure 3.8*.

Note: If after installation, selenium is not available in the Python interpreter, then run the “**pip install selenium**” command in the PyCharm terminal.

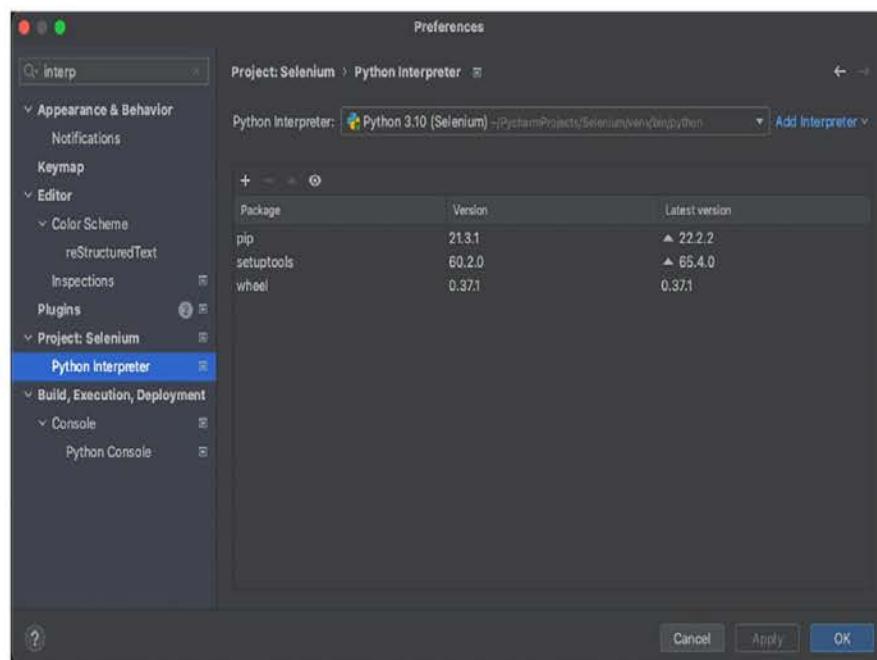


Figure 3.8: Verifying selenium is not downloaded using the Python interpreter

- Verify that Selenium is installed successfully using the Python interpreter, as shown in *Figure 3.9*, or by using pip command “**pip list**”:

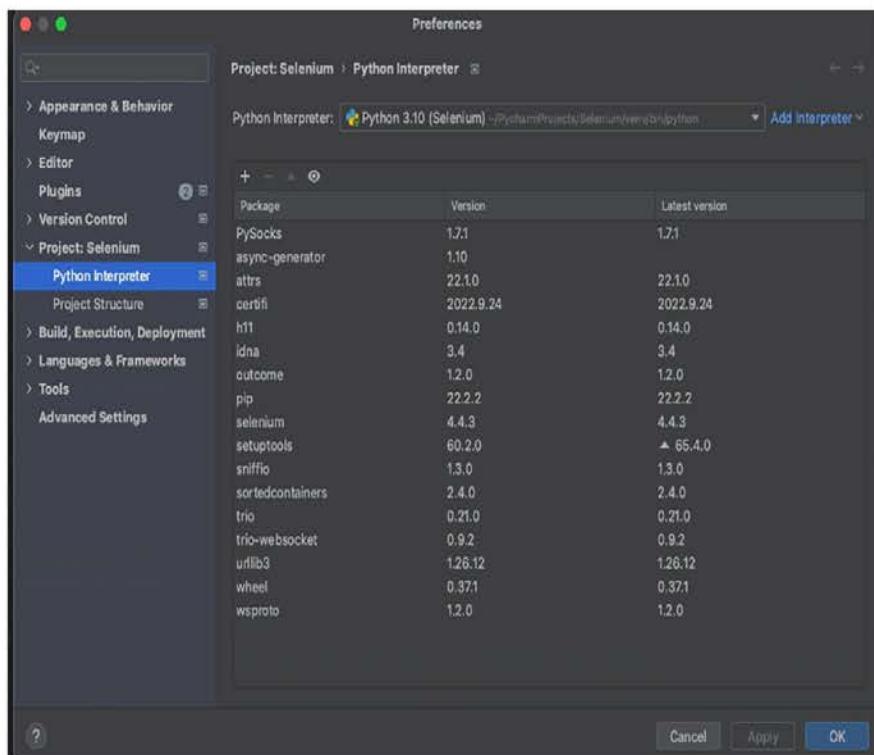


Figure 3.9: Verifying selenium is installed successfully using the Python interpreter

- The link <https://pypi.org/project/selenium/> in step 2 also had the driver location information when scrolling down, as in *Figure 3.10*:

Drivers

Selenium requires a driver to interface with the chosen browser. Firefox, for example, requires [geckodriver](#), which needs to be installed before the below examples can be run. Make sure it's in your PATH, e.g., place it in `/usr/bin` or `/usr/local/bin`.

Failure to observe this step will give you an error `selenium.common.exceptions.WebDriverException: Message: 'geckodriver' executable needs to be in PATH.`

Other supported browsers will have their own drivers available. Links to some of the more popular browser drivers follow.

Chrome:	https://chromedriver.chromium.org/downloads
Edge:	https://developer.microsoft.com/en-us/microsoft-edge/tools/webdriver/
Firefox:	https://github.com/mozilla/geckodriver/releases
Safari:	https://webkit.org/blog/6900/webdriver-support-in-safari-10/

Figure 3.10: Selenium Web drivers link

- Let us download the ChromeDriver. Verify the version of Chrome installed on the machine and download the ChromeDriver of the same version. Unzip and store the executable file in a specific location. Similarly, unzip and download the rest of the drivers as in *Figure 3.11*. Try to open and update security preferences to allow the opening of these drivers.

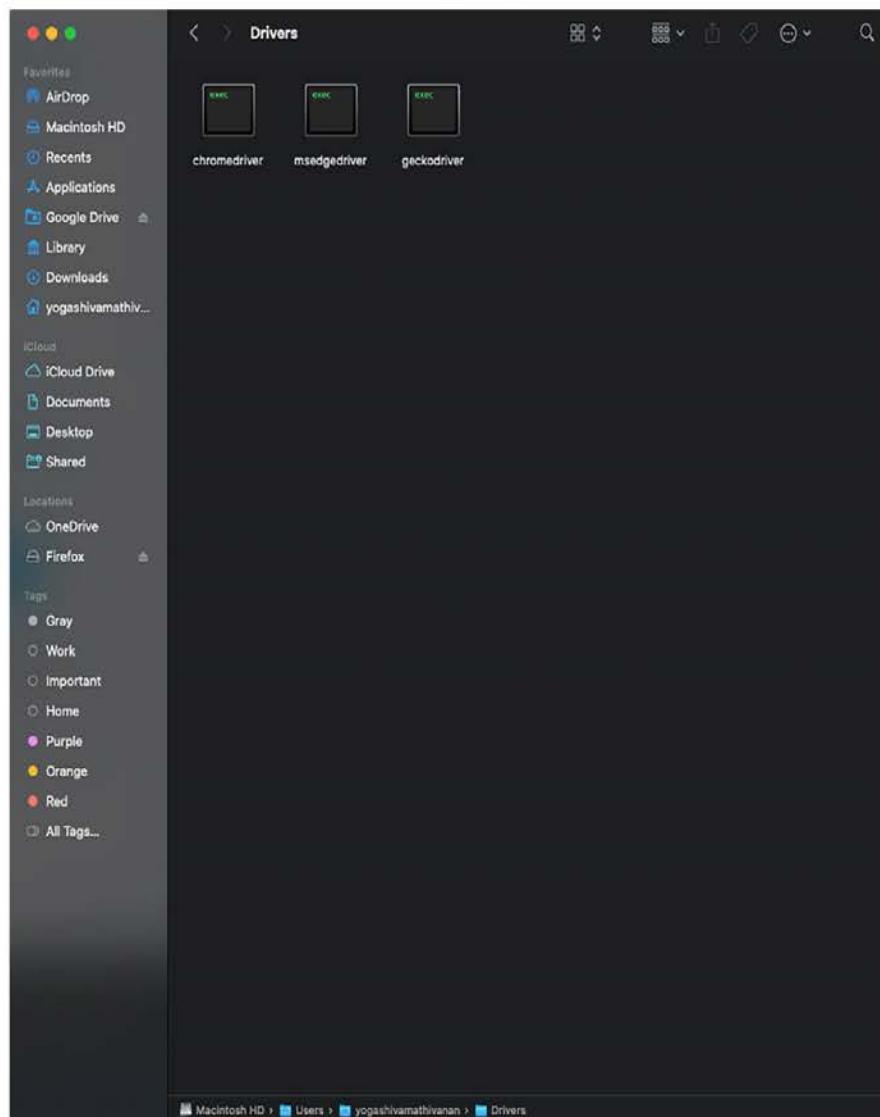


Figure 3.11: Selenium Web drivers downloaded in local machine]

Basic Code: Browser Invoke and Windows setup

As we have downloaded the browser web drivers for Chrome, Firefox, and Edge, we can now write the script and invoke these browsers.

The code for Selenium 3 to invoke the Chrome browser is given as follows. This requires the import of the `webdriver` module from the installed Selenium package with the statement “`from selenium import webdriver`”, followed by passing the `executable_path` variable as the absolute location path of Chrome. The URL of the application is passed as an argument to the webdriver `get()` function, as shown in the following code. Even though there will not be any error during compiling, and you can execute the following code to invoke the Chrome browser, there will be a “`DeprecationWarning`” in the console. Hence, we must use Selenium 4 functions instead.

CODE

```
1. from selenium import webdriver  
  
2. driver = webdriver.Chrome(executable_path="/Users/  
yogashivamathivanan/Drivers/chromedriver")  
  
3. driver.get("https://www.saucedemo.com/")
```

OUTPUT

```
1. DeprecationWarning: executable_path has been deprecated, please  
pass in a Service object  
  
2. driver = webdriver.Chrome(executable_path="/Users/yogashi-  
vamathivanan/Drivers/chromedriver")
```

To invoke the chrome browser, Selenium 4 code is given in the following code. Selenium 4 demands the creation of a service object. “`service_object`” is created by passing the executable path to the Service class and then passing the service object to chrome `webdriver`. This code invokes the browser and opens the saucedemo web application.

CODE

```
1. from selenium import webdriver
```

```
2. from selenium.webdriver.chrome.service import Service
```

```
3. service_object = Service(executable_path="/Users/  
yogashivamathivanan/Drivers/chromedriver")  
4. driver = webdriver.Chrome(service=service_object)  
5. driver.get("https://www.saucedemo.com/")
```

The code structure for Windows is similar as well, and can be seen as follows:

CODE

```
1. from selenium import webdriver  
2. from selenium.webdriver.chrome.service import Service  
  
3. service_obj = Service("C:\chromedriver.exe")  
4. driver = webdriver.Chrome(service=service_obj)  
5. driver.get("https://www.google.com")
```

Invoking the Firefox browser is done using the following code:

CODE

```
1. from selenium import webdriver  
2. from selenium.webdriver.chrome.service import Service  
  
3. service_object = Service(executable_path="/Users/  
yogashivamathivanan/Drivers/geckodriver")  
4. driver = webdriver.Firefox(service=service_object)  
5. driver.get("https://www.saucedemo.com/")
```

Invoking Edge browser is done using the following code:

CODE

```
1. from selenium import webdriver  
2. from selenium.webdriver.chrome.service import Service  
  
3. service_object = Service(executable_path="/Users/  
yogashivamathivanan/Drivers/msedgedriver")  
4. driver = webdriver.Edge(service=service_object)  
5. driver.get("https://www.saucedemo.com/")
```

Browser window setup

There are WebDriver functions that allow modifying the properties of the WebDriver invoked browser window, usually before performing the actions. Here are some examples:

- **maximize_window()** and **minimize_window()**: The current window that the WebDriver is using, is maximized to fit the screen or minimized respectively.
Syntax: `driver.maximize_window(), dirver.minimize_window()`
- **set_window_position()**: Sets the x and y coordinates of the current window.
Syntax: `driver.set_wndow_position(x, y, windowHandle='current')`
- **Refresh()**: Refreshes the current page.
Syntax: `driver.refresh()`
- **Set_window_size()**: Sets the x length and y width of the current webdriver browser window.
Syntax: `driver.set_window_size()`
- **Set_page_load_timeout(wait_time)**: Sets the amount of time to wait for page load to complete before the error. The timeout is in seconds.
Syntax: `driver.set_page_load_timeout(30)`
- **Title**: Gives the current page title.
Syntax: `driver.title`

The following code implements these Window functions:

CODE

1. `from selenium import webdriver`
2. `from selenium.webdriver.chrome.service import Service`
3. `service_object = Service(executable_path="/Users/yogashivamathivanan/Drivers/chromedriver")`
4. `driver = webdriver.Chrome(service=service_object)`
5. `driver.get("https://www.saucedemo.com/")`

```
6. driver.minimize_window()  
7. driver.maximize_window()  
8. driver.refresh()  
9. driver.set_page_load_timeout(30)
```

```
10. driver.set_window_size(500, 500)  
11. print(driver.get_window_size())  
12. print(driver.get_window_position())  
13. print(driver.title)
```

OUTPUT

```
1. {'width': 500, 'height': 500}  
2. {'x': 0, 'y': 25}  
3. Swag Labs
```

First automation: Login scenario

Let us automate a login scenario. The web application we will be using to automate the test is <https://www.saucedemo.com/>. Following are the login scenario test steps we want to automate:

1. Open web browser (Chrome/Firefox/Edge).
2. Open URL <https://www.saucedemo.com/>
3. Provide email: **standard_user**
4. Provide password: **secret_sauce**
5. Click on the **LOGIN** button.
6. Verify the title.
7. Close Browser.

Deprecated Selenium 3 code

Since we have installed Selenium 4, the following code is giving an error in the response. We will see how the same code can be written using Selenium 4 steps.

CODE

```
1. from selenium import webdriver  
  
2. driver = webdriver.Chrome(executable_path="/Users/
```

```
yogashivamathivanan/Drivers/chromedriver")
3. driver.get("https://www.saucedemo.com/")
4. driver.find_element_by_name("email").send_keys("test.account19@
gmail.com")
```

76 ■ Selenium and Appium with Python

```
5. driver.find_element_by_id("loginFormPasswordInput").send_
keys("testadmin")
6. driver.find_element_by_id("loginFormSubmitButton").click()
```

OUTPUT

```
1. DeprecationWarning: executable_path has been deprecated, please
   pass in a Service object
2. driver = webdriver.Chrome(executable_path="/Users/
yogashivamathivanan/Drivers/chromedriver")
3. Traceback (most recent call last):
4. driver.find_element_by_name("email").send_keys("test.account19@
gmail.com")
5. AttributeError: 'WebDriver' object has no attribute 'find_element_
by_name'
```

Selenium 4 test script

CODE

```
1. from selenium import webdriver
2. from selenium.webdriver.chrome.service import Service
3. from selenium.webdriver.common.by import By
4. service_object = Service(executable_path="/Users/
yogashivamathivanan/Drivers/chromedriver")
5. driver = webdriver.Chrome(service=service_object)
6. driver.get("https://www.saucedemo.com/")
7. driver.maximize_window()
8. driver.refresh()
```

```
9. driver.set_page_load_timeout(30)
10. print(driver.get_window_size())
11. print(driver.get_window_position())
12. driver.find_element(By.NAME, "user-name").send_keys("standard_
    user")
13. driver.find_element(By.ID, "password").send_keys("secret_sauce")
```

```
14. driver.find_element(By.ID, "login-button").click()
15. print(driver.title)
16. driver.close()
17. driver.quit()
```

OUTPUT

```
1. {'width': 3440, 'height': 1095}
2. {'x': 0, 'y': 25}
3. Swag Labs
```

Difference between `driver.close()` and `driver.quit()`

In the previous example, in Code line 17 and 18, there is both `driver.close()` and `driver.quit()` at the end of the script. Though during the execution, using either of them closes the browser but `driver.close()` function is used to close the current window, and `driver.quit()` is used to terminate the entire execution of the driver, closing every associated window.

Webdriver Manager

Python library is used to automatically manage browser drivers without having to download and provide the path. For now, it supports ChromeDriver, GeckoDriver, IEDriver, OperaDriver, and EdgeChromiumDriver. So far, we had to download the chromedriver, unzip and save it in a specific folder, then set the path to the driver.

Before webdriver manager in Selenium 3

```
1. from selenium import webdriver
2. driver = webdriver.Chrome(executable_path="/Users/yogashi-
    vamathivanan/Drivers/chromedriver")
```

Before webdriver manager in Selenium 4

```
1. from selenium import webdriver  
2. from selenium.webdriver.chrome.service import Service  
3. service_object = Service(executable_path="/Users/  
yogashivamathivanan/Drivers/chromedriver")
```

Install Webdriver-manager using the command “`pip install webdriver-manager`”. Following is the code after installation, to invoke the chrome driver.

In the following code, when `chromedrivermanager` is called, it will download the latest version of Chrome and execute the code. We can also pass the version of the Chrome we want to run the script on, using syntax, `driver = webdriver.Chrome(ChromeDriverManager("100.0.4896.20").install())`,

However, this requires that the provided Chrome version should map with the Chrome version in your machine.

After webdriver manager in Selenium 3

```
1. from selenium import webdriver  
2. from webdriver_manager.chrome import ChromeDriverManager  
  
3. driver = webdriver.Chrome(ChromeDriverManager().install())  
4. driver.get("https://www.google.com")
```

After webdriver manager in Selenium 4

```
1. from selenium import webdriver  
2. from selenium.webdriver.chrome.service import Service  
3. from webdriver_manager.chrome import ChromeDriverManager  
4. driver = webdriver.Chrome(service=Service(ChromeDriverManager().  
install()))
```

Webdriver manager versus driver path

As seen in the previous sections, the WebDriver manager is a more convenient way

of managing the webdriver compared to the driver. A more detailed difference is shown in the following *Table 3.2*:

	WebDriver Manager	Driver Path
Definition	A tool that automatically downloads and manages the correct web driver version for the browser being used.	The physical location of the web driver executable file on the local system.

	WebDriver Manager	Driver Path
Installation	Requires installation of the webdriver manager library in the code.	Requires manual download and installation of the web driver executable file for the specific browser.
Configuration	Automatically configures the web driver path based on the browser being used.	Requires explicit configuration of the web driver path in the code.
Maintenance	Automatically updates and manages the web driver versions for the browser being used.	Requires manual updates and management of the web driver versions for the specific browser.
Benefits	Provides convenience and reduces maintenance effort.	Offers greater control and customization options.
Limitations	May not support all web driver versions or browsers.	Requires manual installation and configuration for each browser and operating system.

Table 3.2: Differences between WebDriver manager and Driver Path

Conclusion

Python with Selenium is the most popular automation testing combination. Selenium 4 comes with new features to close the gaps and performance, compared to the previous version. Hence, the popularity of Python + Selenium is going to continue to grow. In this chapter, we have learned in depth the basics and evolution of the selenium tool. The next step in Selenium is to do in-depth concepts of the Selenium, starting with the locators which are essential to work with a variety of web elements in *Chapter 5 Locators and Handling Web and Advanced Web Elements*.

In the next chapter, we will go over a similar process to understand the in-depth basics of APPPIUM for mobile automation. The demand for mobile automation in the modern world is growing exponentially. Therefore, let us take web and mobile automation in parallel.

Key facts

- Selenium is an open-source web application automation tool, that supports all major browsers.

- Selenium Suite consists of four tools, including Selenium IDE, Selenium RC, Selenium Webdriver, and Selenium Grid.
- Four layers of any web application include the presentation layer, data service layer, business logic layer, and data access layer.
- Major difference between Selenium 3 and Selenium 4 is communication between client and browser driver. Where selenium 3 communicates using JSON Wire protocol, Selenium 4 communicates directly.
- Install Selenium using pip command `pip install selenium`. Validate the installation in the Python interpreter in pycharm preference or settings.
- `Driver.close()` closes the current window. `driver.quit()` closes all the driver execution.
- Webdriver manager allows installation of WebDriver in real time during execution without having to download, store and pass the path during execution.

Questions

1. What are the advantages of Selenium?
2. What are the different components of Selenium 4 architecture?
3. What is the library to import to invoke?
4. What driver function should we use to maximize the execution window and refresh the current page?
5. How to find an element in Selenium 4 compared to Selenium 3? What error occurs when you use Selenium 3 command in Selenium 4?
6. What is service object?

7. What is WebDriver manager?

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



CHAPTER 4

Appium for Mobile Automation

Introduction

Appium is an open-source cross-platform tool for automating native, mobile web apps and hybrid applications on Android mobile, IOS mobile, and Windows desktop platforms developed by Sauce Labs. Mobile app benefits are growing extensively, varying from communications to bookings to banking services done on the mobile phone. Today, nearly 91% of the world's population owns a phone. The software

Industries have embraced a mobile-first strategy given the evolving mobile app usage market; this made the testing process more challenging in terms of covering an exhaustive list of models. Mobile automation engineers are in demand more than ever, to automate native apps, mobile web apps, and hybrid apps on mobile. Appium by far is the most popular automation testing tool for mobile apps.

Native apps are those written using the iOS, Android, or Windows SDK, and are developed specifically for one platform. A native app can be installed using the application store (like Google Play and Apple Play Store). Mobile web apps are not real applications; they are websites accessed using a mobile web browser. Appium supports Safari on iOS and Chrome or the built-in 'Browser' app on Android. Hybrid apps are a mixture of web apps and native apps and have a wrapper around a "webview", a native control that enables interaction with web content.

Appium uses Selenium and JSON Wire protocol internally to interact with iOS and Android apps, using Selenium webdriver. Appium also allows running automated test scripts over mobile and tablet devices. In this chapter, we will look at in-depth basics of automation of these different mobile apps using Appium, test setup, internal architecture of Appium, key practices, and other tools.

Structure

In this chapter, we will discuss the following topic:

- Appium tool for Mobile Automation
- Appium Setup on Windows
- Appium Setup on Mac
- Android Emulator Configuration on Mac and Windows
- Application Launching: Android
- iOS Simulator configuration on Mac

Objectives

This chapter will explain the Appium tool for mobile web, native, and hybrid app automation, using Python as the programming language. We will start with Appium architecture, advantages of Appium compared to other tools, installation, and setup of Appium and other tools, as well as basic code to invoke and start the automation testing. We will apply Python and Selenium learning to learn how to implement the **Appium Python library** and **Selenium webdriver** in mobile automation.

Appium tool for mobile automation

Appium is the most popular, widely-used open source, and cross-platform mobile automation tool, carrying out automation testing of native, hybrid, and mobile web applications.

Mobile application testing

Mobile testing is undertaken by QA engineers, by using tools to interact with the part of the application impacted by the code change. QA engineers usually use one of the following three options:

- **Real Mobile Device:** QA can test the app directly on the test mobile device and replicate the end-user experience. This requires manual efforts and is time-consuming.

- **Cloud-based Real device:** QA can test the application pre-installed in OS and browsers and run on the device in a remote testing environment. Testing can be done on multi-devices with different configurations and versions, and it also allows QA engineers to manage these devices remotely.
- **Android Emulator/ IOS Simulator:** This is a simulation of a real mobile device with software configuration on a desktop.

Types of Mobile Testing

There are certain types of needs to be considered with mobile applications, apart from functional testing. They are as follows:

- **Compatibility testing:** The mobile application's behavior on multiple platforms is tested, such as testing the application compatibility with mobile, tablets, TV, desktops, and so on.
- **Usability testing:** This is to test the complete UX of an application, where interaction experience with the mobile application is tested, actions like a tap, hold, and scrolling.
- **Performance testing:** Key performance metrics such as Load, and stress on the application are tested and assessed to analyze the behavior such as response time, and errors, of the application under high user load.
- **Interrupt testing:** This is more common mobile-centric testing that is done to analyze the behavior of the application during interruptions such as calls and messages, and switch to other applications and switch back.
- **Security testing:** Application vulnerabilities to cyber threats are tested to

- **Memory optimization testing:** Test the optimum mobile memory usage of the application.

Appium architecture

Appium is an HTTP server written in Node.js. The server operation is executed using Rest API in the order of, firstly receiving connection from the client and initiating a session; secondly, listening and executing the command, and finally returning the execution response. Thus, Appium is a client-server architecture.

Appium server receives JSON object connection requests over HTTP through JSON-Wire Protocol. A session id is generated, under which the testing is performed. *Figure 4.1* shows the high-level architecture of Appium:

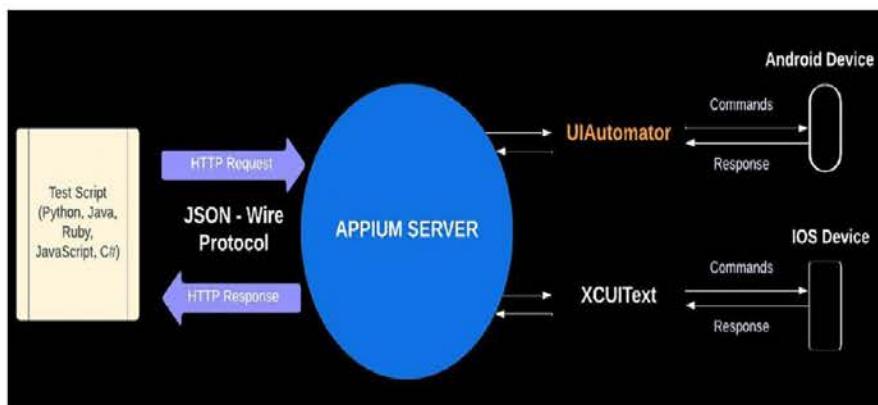


Figure 4.1: Appium Architecture

Appium consists of four stages of functions, as shown in the preceding *Figure 4.1*:

- **Appium Client Library:** The automation code in a supported programming language like Python, Java, JavaScript, Ruby, C#, and so on. This test script is converted to JSON format.
- **JSON Wire Protocol:** The converted JSON format is transferred to the Appium server over HTTP protocol to handle the request.
- **Appium Server:** The Appium server built on top of Node.js is a wrapper that translates Selenium WebDriver commands into XCUItest for iOS devices and UIAutomator/UIAutomator2 for automation on Android devices or emulators. It validates the platform device and application under test details and then communicates with UIAutomator or XCUItest to perform the script actions.
- **UIAutomator2/ XCUItest/WinAppDriver:**
 - o UIAutomator2 is the latest version developed by Google, which is the test framework to test native app UI. Appium uses the **appium-android-bootstrap** module to interact with UIAutomator2, which processes the request from the Appium server and transfers the request to an Android device or emulator. When the Appium client requests to create a new “AndroidDriver” session, the desired capability is passed by the client to the Appium node server. Then, the UIAutomator2 module creates a session, installs the UIAutomator2 server APK on the connected device, starts the Netty server, and initiates a session. Once the Netty server session starts, the UIAutomator2 server continues to listen on the device for requests and provide expected responses.
 - o XCUItest is an automation framework introduced by Apple, which is used to develop UI tests for iOS apps. Appium internally uses

"WebDriverAgent" provided by Facebook, which is an iOS WebDriver server used to remotely control a connected iOS device or a simulator. It allows it to launch the app and run commands like tap/scroll and to any application. Appium uses Apple's UIAutomation library for the older version, which communicates with bootstrap.js running inside the simulator to perform the commands received.

- o WinAppDriver is used for Windows PC desktop apps.

Appium drivers

The drivers allow Appium to automate multiple platforms. To run the tests on a real device, the device needs to be installed with the Appium driver.

A list of Drivers is as follows:

- XCUITest: iOS 9.3 and above
- UIAutomation: iOS 9.3 and lower
- UIAutomator/UIAutomator2: Android 4.3 and above
- Espresso: Android
- Selendroid: Android
- WinAppDriver: Windows

Advantages of Appium

The advantages of Appium are as follows:

- **Open Source:** Appium can be utilized free of cost.
- **Easy to understand:** As Appium is built on selenium, with all selenium features available, Selenium engineers can easily adapt the Appium automation tool. New users without Selenium knowledge also have a leaner learning curve compared to other tools.
- **Community support:** Appium support has been the tool's major benefit. The support comprises a large community of contributors to keep the users updated and resolve issues.
- **Multi programming language support:** Appium supports multiple scripting languages such as Java, Python, JavaScript, and so on, that allows users to select a scripting language of choice for automation
- **Cross-platform automation:** Appium supports automation of both Android and iOS, which are majorly used operating systems with the same

code. Appium can be integrated with emulators, simulators, and cloud environments, which is useful in optimizing execution time.

- **Appium inspector:** This provides support for record and playback like selenium IDE. Using the user actions on the native apps can be recorded and converted into a selected script.

Appium setup on Windows

We will set up the prerequisites, and install the Appium tool, and supporting tools. Follow the given steps for complete installation of Appium on a Windows machine:

1. Install Python. Follow the installation steps from *Chapter 2, Python programming Basics, Installation, and Environment Setup*.
2. Install PyCharm. Follow the installation steps from *Chapter 2, Python programming Basics, Installation, and Environment Setup*.
3. **Appium and Selenium Python Libraries:**
 - a. Navigate to page <https://pypi.org/project/Appium-Python-Client/> and copy the pip installation command, as shown in *Figure 4.2*. Paste it into the command terminal.

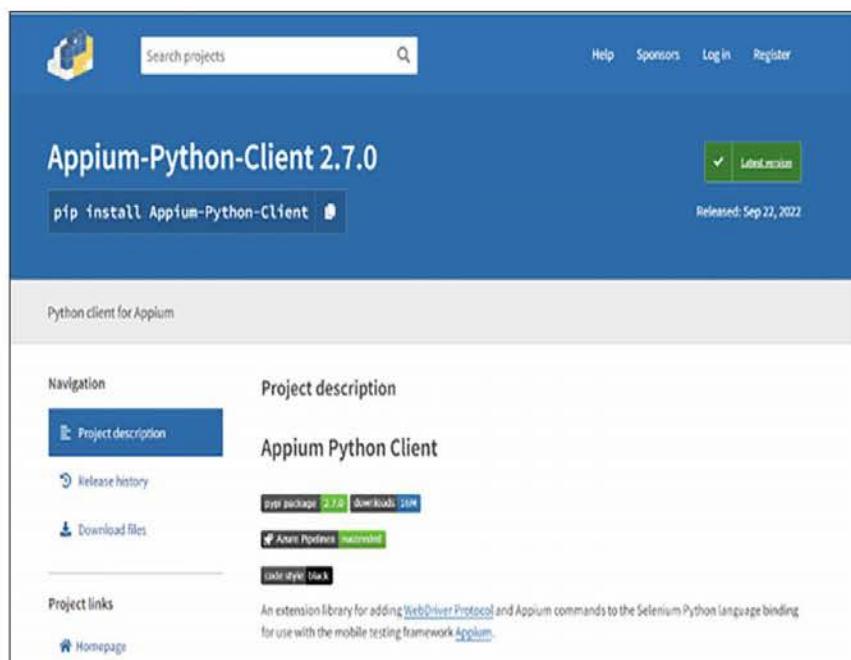


Figure 4.2: Latest Appium Python Client

- b. Open PyCharm, navigate to **Settings | Interceptor**, and validate the installation of Appium-Python-client along Selenium automatically if not installed already. Refer to *Figure 4.3*:

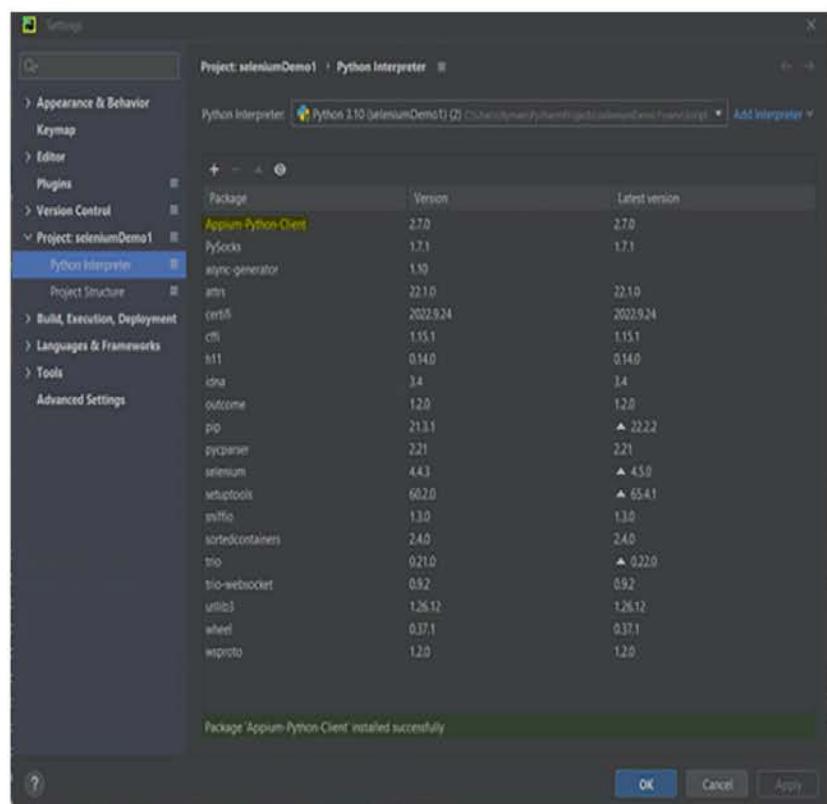


Figure 4.3: PyCharm Interceptor Appium Python Client Installed

4. Android Studio Installation:

- Navigate to <https://developer.android.com/studio> and click on **Download Android Studio**, as shown in Figure 4.4.

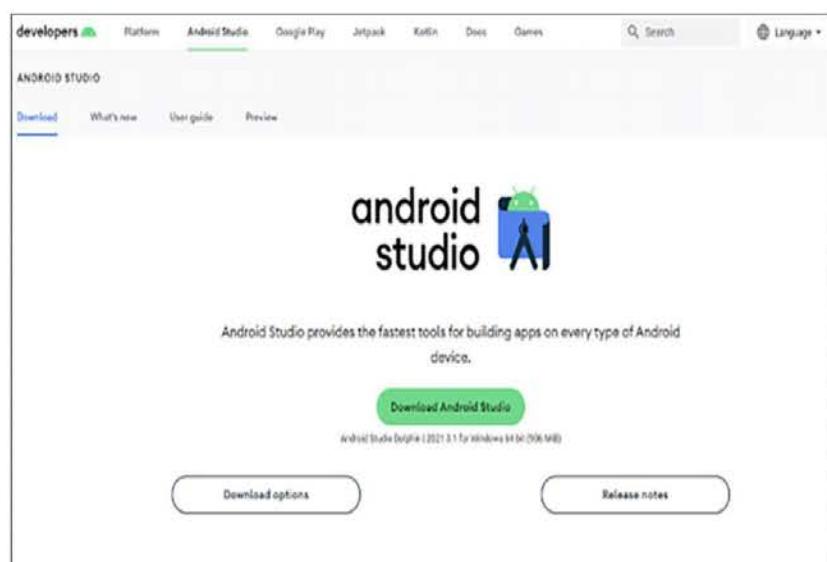


Figure 4.4: Android Studio download location

- b. After the installation is complete, open the Android application and choose the option “**Do not import settings**” and complete the installation with default settings.
- c. Once Completed, click finish and open “**New Project**”. Select a basic activity template and provide a name as shown in *Figure 4.5* and click on **Finish**. It will take some time to set up. Navigate to **Tools | Android SDK**, where we can install the Android version we require. Also, make a note of the Android SDK location.

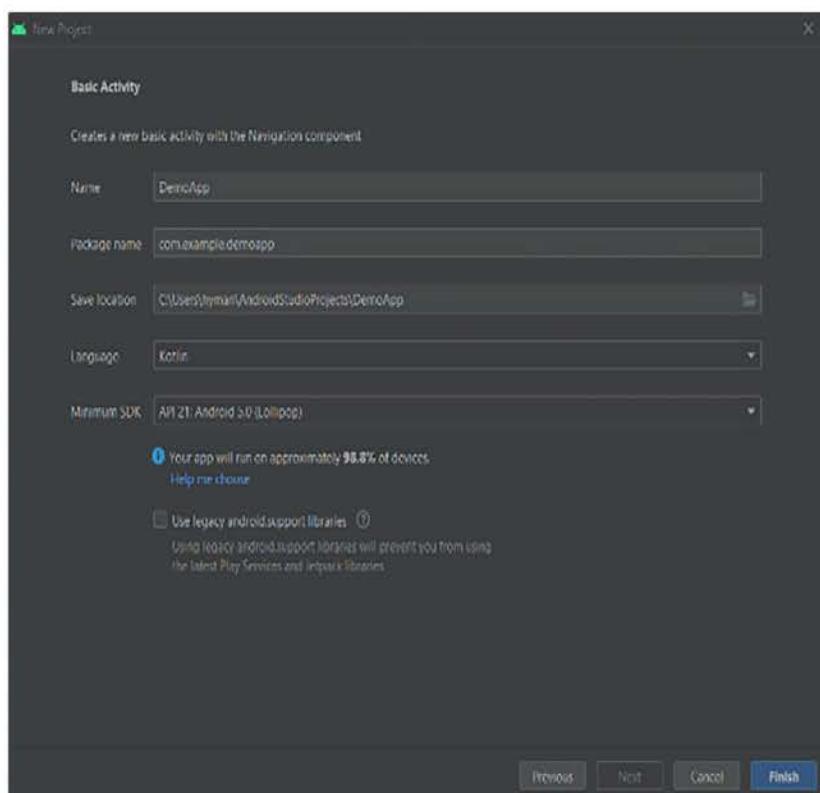


Figure 4.5: Android Studio New Project

- d. Now we need to set the system variable environment variable **ANDROID_HOME** to the SDK location verified in the last step, as shown in *Figure 4.6*.

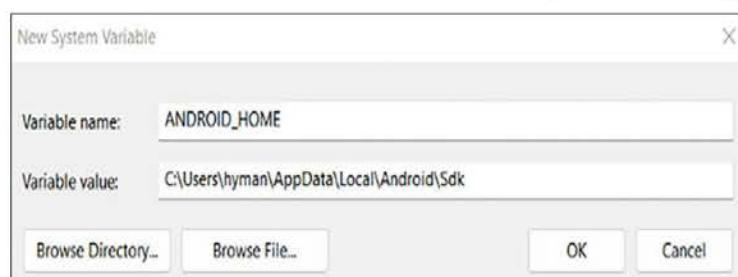


Figure 4.6: ANDROID_HOME Environment Variables

- e. Also set the path variables to platforms and platform tools, as shown in *Figure 4.7*:

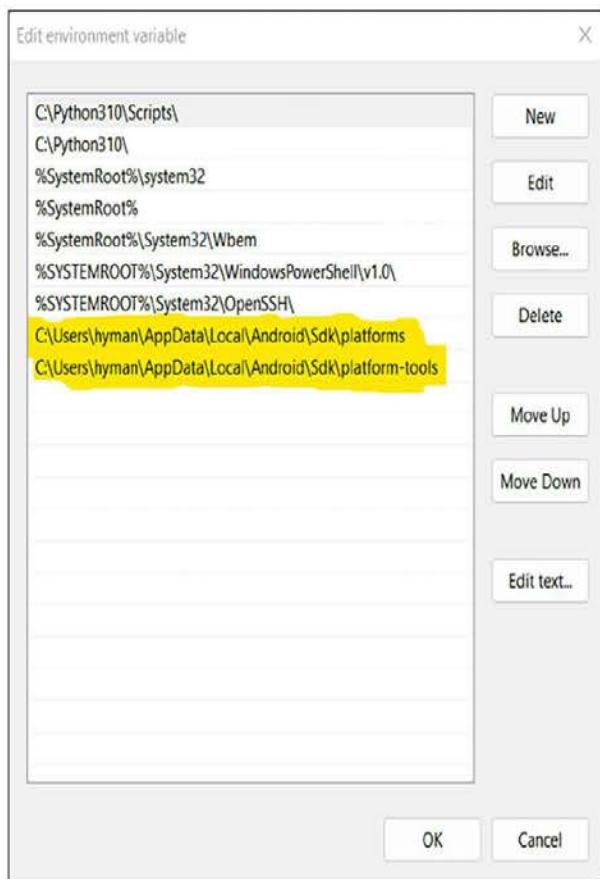


Figure 4.7: Android Studio Path system variable

- f. To verify everything was set up as expected, open the command prompt and type “**adb drivers**”, which should return “List of devices attached”. This command displays the list of devices connected when we use the real device.
5. **Appium server app installation:** Once the libraries are installed, let us install the desktop App. Navigate to <https://appium.io/downloads.html> and click on the Appium-Desktop link, as shown in *Figure 4.8* and download the .exe file and follow the instructions for installation. Once the installer is opened, select the “**Anyone who uses this computer**” option so every user on the machine can access it. After installation validates, the application can be opened and we can start the server at the default port 4723.

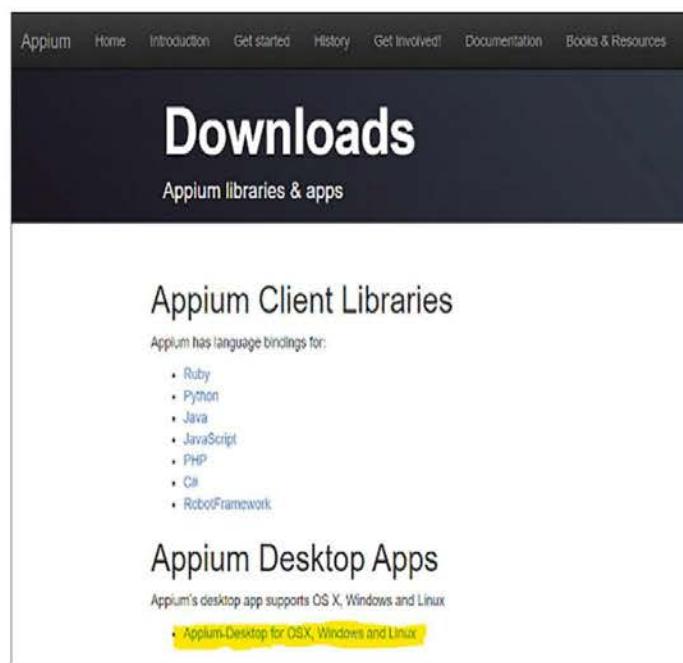


Figure 4.8: Appium Desktop Server app download

6. **Appium Inspector Installation:** Web search Appium inspector or navigate to GitHub <https://github.com/appium/appium-inspector>. Scroll to the installation section, click on **Releases**, and install the .exe file. Open the installed application, as shown in *Figure 4.9*:

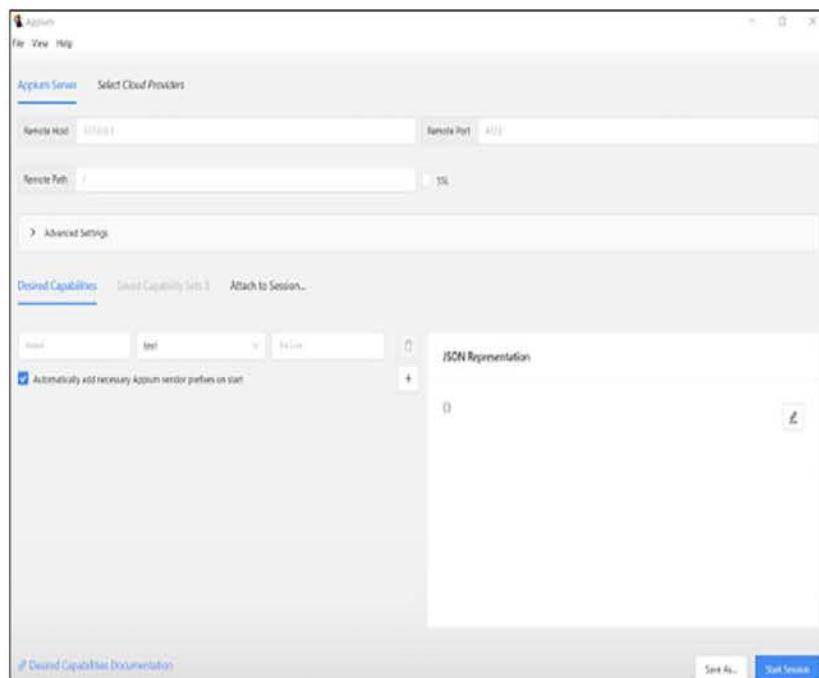


Figure 4.9: Appium Inspector application

7. **Java Installation:** Appium also requires that Java be installed and environment variables are set. Navigate to <https://www.oracle.com/java/technologies/downloads/#java17> to install Java 17 JDK, Appium requires Java 8 or greater. Download the JRE from <https://www.oracle.com/java/technologies/javase/javase8-archive-downloads.html>. Now set the environment variables, as shown in *Figure 4.10*:

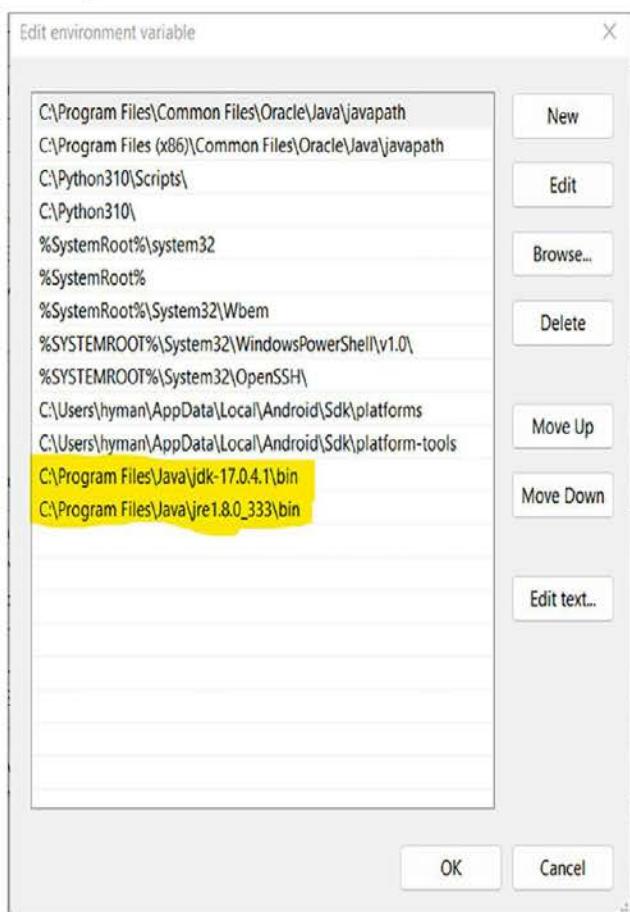
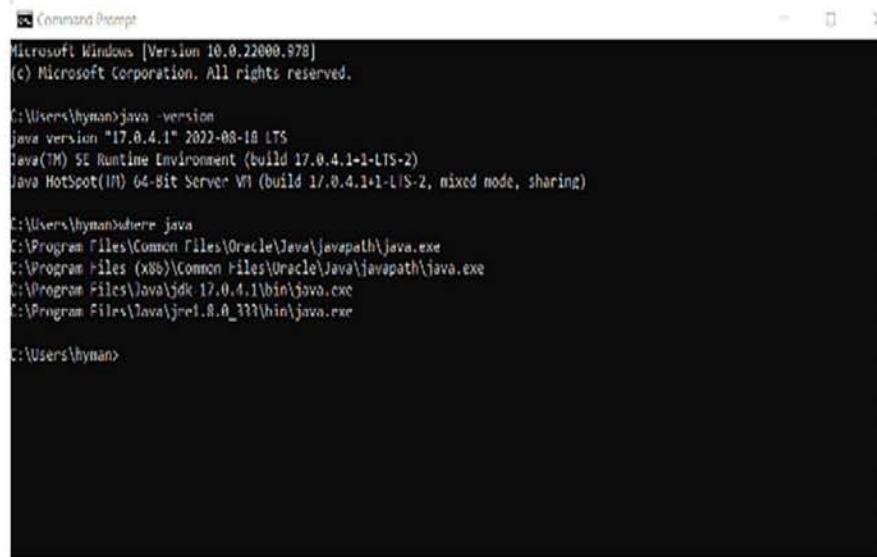


Figure 4.10: Java System Environment Variable

- a. Once the Installation is completed and environment variables are set, open the command prompt, and verify the installation by using the command “`java -version`” and “`where java`”, as shown in *Figure 4.11*:



```
Command Prompt
Microsoft Windows [Version 10.0.22000.978]
(c) Microsoft Corporation. All rights reserved.

C:\Users\hyman>java -version
java version "17.0.4.1" 2022-08-18 LTS
Java(TM) SE Runtime Environment (build 17.0.4.1+1-LTS-2)
Java HotSpot(TM) 64-Bit Server VM (build 17.0.4.1+1-LTS-2, mixed mode, sharing)

C:\Users\hyman>where java
C:\Program Files\Common Files\Oracle\Java\javapath\java.exe
C:\Program Files (x86)\Common Files\Oracle\Java\javapath\java.exe
C:\Program Files\Java\jdk_17.0.4.1\bin\java.exe
C:\Program Files\Java\jre1.8.0_333\bin\java.exe

C:\Users\hyman>
```

Figure 4.11: Java System Environment Variable

- Open the Appium server application installed in step 5 and click on **Edit configuration** and add **ANDROID_HOME** and **JAVA_HOME** Path. Click on “Save and Restart”, as shown in *Figure 4.12*.

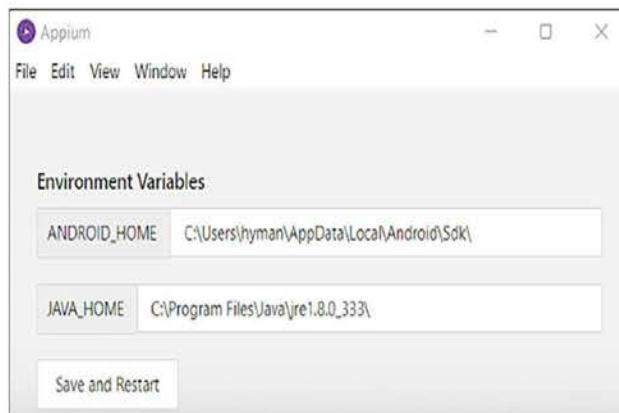


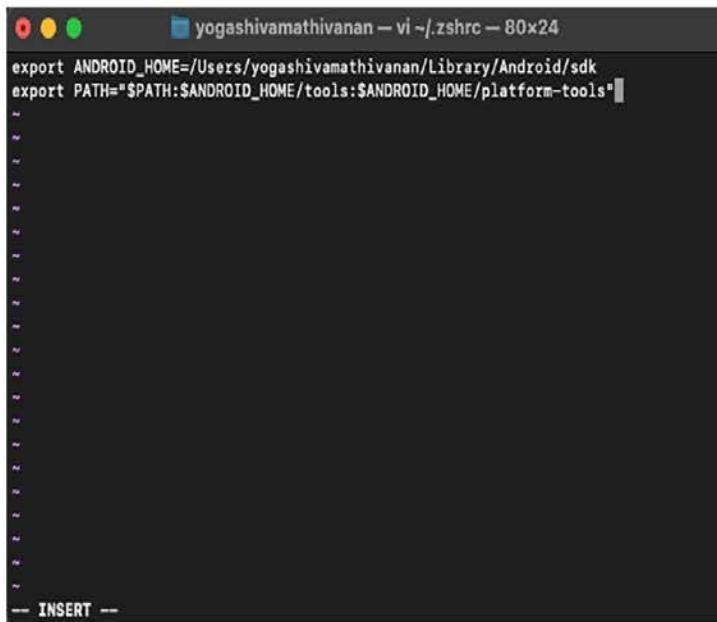
Figure 4.12: Java System Environment Variable

Appium setup on Mac

The Setup installation of Appium and tools is similar on Mac compared to Windows. The major difference is in the XCode tool which can only be installed on Mac for testing iOS devices. We will discuss the XCode and the differences in the installation process on Mac. Refer to the following:

- Install Python:** Follow the installation steps from *Chapter 2, Python programming Basics, Installation, and Environment Setup*.

2. **Install PyCharm:** Follow installation instructions from *Chapter 2, Python programming Basics, Installation, and Environment Setup*.
3. **Appium and Selenium Python libraries:** Similar to the steps for Windows.
4. **Android studio installation:** Navigate to <https://developer.android.com/studio> and click on **Download Android Studio**, as shown in *Figure 4.13*. Then click **Download** on one of the options: **Mac with Apple Chip** or **Mac with Intel Chip**.



The screenshot shows a terminal window titled "yogashivamathivanan — vi ~/.zshrc — 80x24". The window contains the following text:

```
export ANDROID_HOME=/Users/yogashivamathivanan/Library/Android/sdk
export PATH="$PATH:$ANDROID_HOME/tools:$ANDROID_HOME/platform-tools"
```

The bottom right corner of the terminal window displays the text "-- INSERT --".

Figure 4.13: Android Studio Environment Variable

- a. Move the Android Studio to Applications.
- b. Follow the installation steps as in Windows.
- c. Set environment variables. Open the terminal and type "**adb**" to validate that the environment variables are not set.
- d. Open the config file by using the command "**vi ~/.zshrc**" and press "**i**" to insert mode and set the **ANDROID_HOME** variable and path variable, as shown in *Figure 4.13*. Once the config environments are set, press the **Esc** key, then colon "**wq**" (write quit). Then in the command, type "**source ~/.zshrc**".
- e. Now type the "**adb**" command. It should show the result, as in *Figure 4.14*:

```
yogashivamathivanan ~/Users/yogashivamathivanan ~ zsh ~ 239x86
Android Debug Bridge version 1.0.11
Version 33.0.3-6052218
Installed at /Users/yogashivamathivanan/Library/Android/sdk/platform-tools/adb

global options:
  -a          listen on all network interfaces, not just localhost
  -d          use USB device (error if multiple devices connected)
  -e          use TCP/IP device (error if multiple TCP/IP devices available)
  -s SERIAL   use device with given serial identifier $AVD_NAME.$SERIAL
  -t TO       use device with given transport id
  -c          name of adb server host (default localhost)
  -l          port of adb server (default 5037)
  -r          poll for adb server (default 100ms)
  -l SOCKET   listen socket connection for adb server (default tcp:localhost:5037)
--no-service START/USB only allowed with 'start-server' or 'server nohidden', server will only connect to one USB device, specified by a serial number or USB device address.
-eall-on-midi-videos exit if stdout is closed

general commands:
devices [-l]      list connected devices (-l for long output)
help             show this help message
version          show version num

networking:
connect HOST[:PORT]  connect to a device via TCP/IP (default port:5037)
disconnect [-HOST[:PORT]]  disconnect from given TCP/IP device (default port:5037), or all
port HOST[:PORT] [LISTENING PORT]  port forward
reverse [-L] [HOST[:PORT]] [DEVICE]  reverse TCP/IP communication
forward [-L] [HOST[:PORT]] [LOCAL] [REMOTE]
  forward socket connection usage:
  $ adb forward [local] [remote] [localabstract|remote abstract]
```

Figure 4.14: Validation of Android Studio Environment Variable setup

5. **Appium Server App Installation:** Once the libraries are installed, let us install the desktop App. Navigate to <https://appium.io/downloads.html> and click on the Appium-Desktop link, as was shown in *Figure 4.8*. Download the .dmg file and follow the instructions for installation. Once the installer is opened, select the “**Anyone who uses this computer**” option so that every user on the machine can access it. After installation validates, the application can be opened and start the server at the default port 4723.
6. **Appium Inspector Installation:** Web search Appium inspector or navigate to GitHub <https://github.com/appium/appium-inspector>. Scroll to the installation section, click on releases, install the .dmg file, open and move the Appium Inspector app to the application and open. The application will be similar to *Figure 4.9*.
7. **XCode Installation:** The Xcode app can be installed by App Store if it is not already installed.
8. **Java Installation:** Web search Java download, or navigate to the link <https://www.oracle.com/java/technologies/downloads/#java17>, download the latest or desired version of the .dmg file, and complete the installation. Java 17 is the latest LTS, that is, the Long-Term Support version of Java.
 - a. **Set Environment Variables:** Open the terminal and type the command “`/usr/libexec/java_home -v17`” with the version of Java you want to set the environment. The response to the command is the “**Java_Home**” Path. Now type the command “`vi ~/.zshrc`” and export the path from the last command, as shown in *Figure 4.15*. Then click the *Esc* key, colon “`wq`”, which will exit out of the config file. Type the command “`source ~/.zshrc`” to make the commands in the file executable.

This screenshot shows a terminal window titled 'yogashivamathivanan — vi ~/zshrc — 92x30'. The window displays the contents of a shell configuration file ('zshrc') with the following code:

```
export ANDROID_HOME=/Users/yogashivamathivanan/Library/Android/sdk
export PATH="$PATH:$ANDROID_HOME/tools:$ANDROID_HOME/platform-tools"
export JAVA_HOME=/Library/Java/JavaVirtualMachines/jdk-17.0.4.1.jdk/Contents/Home
```

The cursor is positioned at the end of the third line, indicated by a vertical bar.

Figure 4.15: Java Environment Variable set up in Mac

9. Java installation will then be complete, as shown in *Figure 4.16*:

This screenshot shows a terminal window titled 'yogashivamathivanan — -zsh — 92x30'. The window displays the results of running the Java command ('java --version') after sourcing the configuration file. The output is:

```
[yogashivamathivanan@Yogashivas-MacBook-Pro ~ % source ~/zshrc
[yogashivamathivanan@Yogashivas-MacBook-Pro ~ % java --version
java 17.0.4.1 2022-08-18 LTS
Java(TM) SE Runtime Environment (build 17.0.4.1+1-LTS-2)
Java HotSpot(TM) 64-Bit Server VM (build 17.0.4.1+1-LTS-2, mixed mode, sharing)
yogashivamathivanan@Yogashivas-MacBook-Pro ~ %]
```

Figure 4.16: Java Installation Complete

Android Emulator configuration on Mac and Windows

We have gone over the installation of Android studio. Let us now look at the Emulator part of it. Follow the given steps to set up the Android Emulator. The steps for Mac and Windows are quite similar.

1. Navigate to **Tools** and click on **Device manager** or **AVD Manager** (depending upon version) and select the phone option and device that you want to configure the emulator, as in *Figure 4.17*.

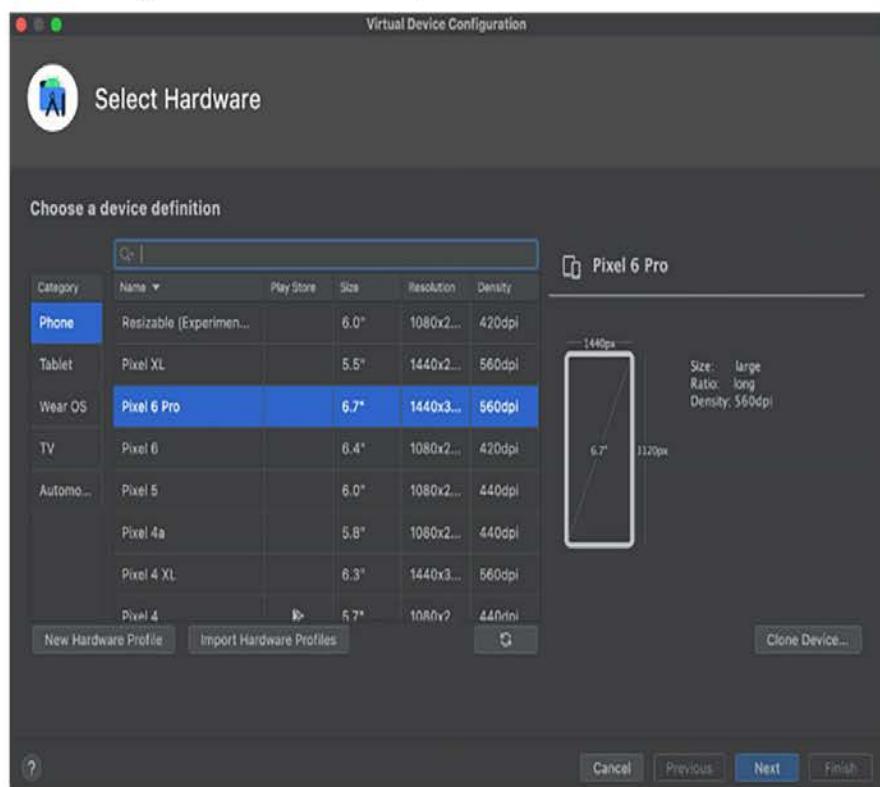


Figure 4.17: Android Emulator Device Manager

2. Then click on **Next** and select the latest or preferred Android version/API, as shown in *Figure 4.18*. The download will take some time, and then provide a name to the device and click on **Finish**. Please refer to the following figure:

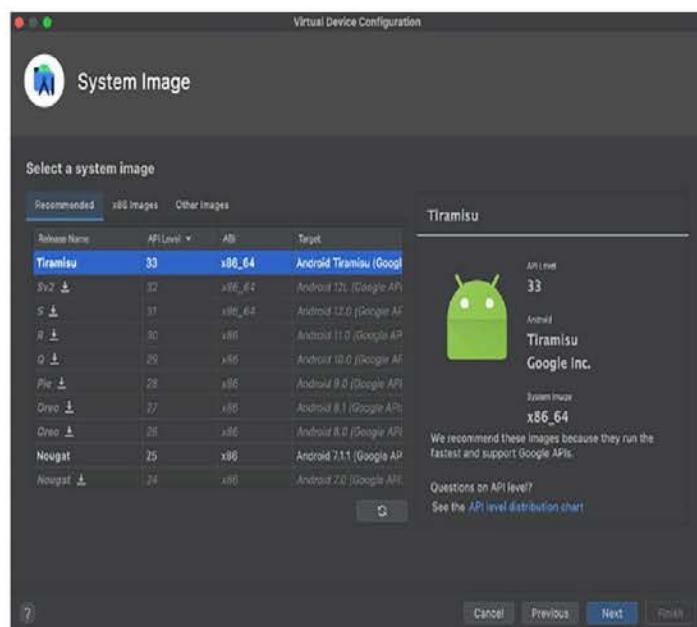


Figure 4.18: Android Emulator Device API

3. Navigate to Device Manager again to see how the new device should be showing up. Then click on the **Play** button next to the device, which will bring the emulator up and running as in *Figure 4.19*.

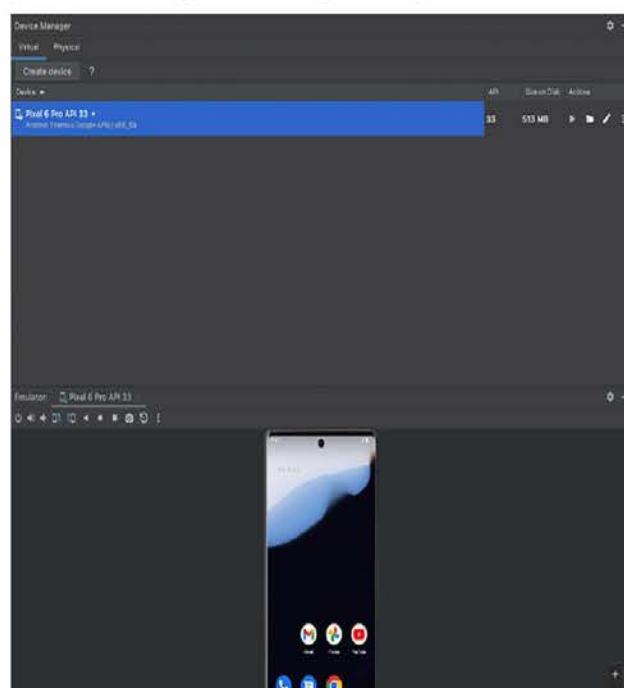
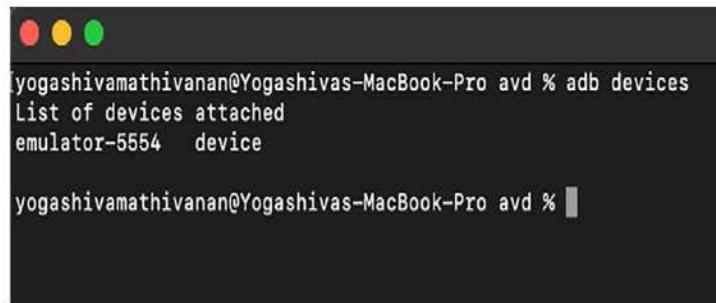




Figure 4.19: Android Emulator Device

4. Navigate to the terminal and run the command “**adb devices**”. The new emulator should show up as in *Figure 4.20*:



```
yogashivamathivanan@Yogashivas-MacBook-Pro ~ % adb devices
List of devices attached
emulator-5554    device

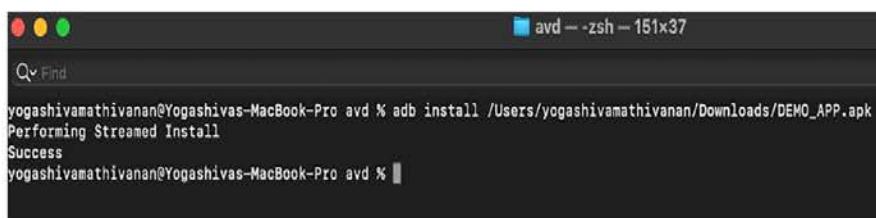
yogashivamathivanan@Yogashivas-MacBook-Pro ~ %
```

Figure 4.20: Android Emulator Device Validation in Terminal

Installation of APK File

APK stands for **Android Package** or **Android Package Kit**. It is like the .exe file used to install programs on Windows. .apk file is the format used by Android to install applications.

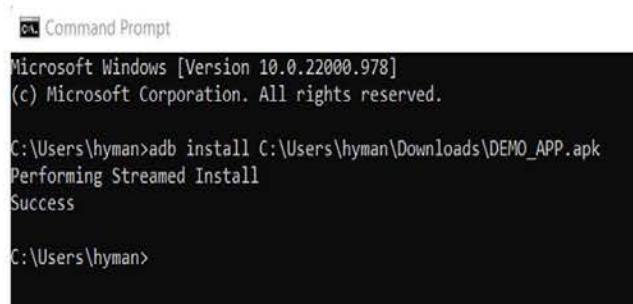
Once we have the apk file of the application under test, this can be installed directly to the emulator using the command **adb install <file path>** as shown in *Figure 4.21*.



```
avd -- zsh - 151x37
yogashivamathivanan@Yogashivas-MacBook-Pro ~ % adb install /Users/yogashivamathivanan/Downloads/DEMO_APP.apk
Performing Streamed Install
Success
yogashivamathivanan@Yogashivas-MacBook-Pro ~ %
```

Figure 4.21:Installing apk file using file path

Instead of providing the path of the file, you can type **adb install**, then drag and drop the apk file to the terminal, as shown in *Figure 4.22*:



```
Microsoft Windows [Version 10.0.22000.978]
(c) Microsoft Corporation. All rights reserved.

C:\Users\hyman>adb install C:\Users\hyman\Downloads\DEMO_APP.apk
Performing Streamed Install
Success

C:\Users\hyman>
```

Figure 4.22: APK Installation without using file path

The application should be installed in the emulator, as shown in *Figure 4.23*:

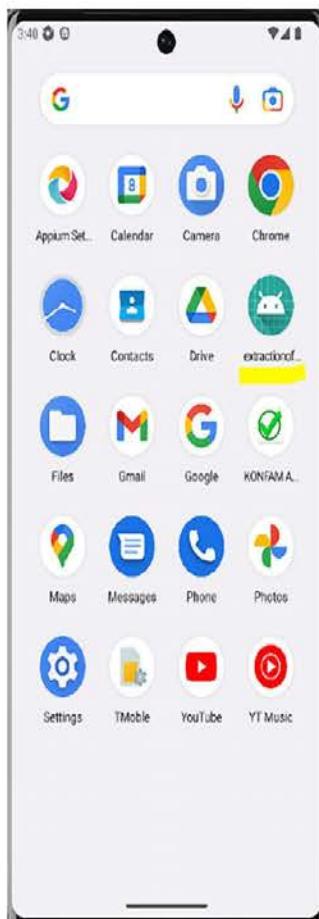


Figure 4.23: Application Installed

Appium Desired Capabilities

Since we have already seen the installation of Appium Inspector in step 6 of the Appium setup above, Desired Capabilities are the way to provide information about the application, platform, device name, and so on.

When a new automation session is requested to the Appium server, navigate to <https://appium.io/docs/en/writing-running-appium/caps/> for a complete list of desired capabilities. The following are some significant ones when working with Android.

- **appPackage:** The package name of the application we want to launch. To retrieve, open the app in the emulator and go to the terminal and type the command `adb shell dumpsys window windows`. This will provide the package name and activity name as in *Figure 4.24* where, package name

is **com.Testing.karthik.extractionofuserid** and activity name is **com.Testing.karthik.extractionofuserid.MainActivity**.

```
Entry TaskId=0
    topApp=ActivityRecord{ee795de} U com.Testing.karthik.extractionofuserid/.MainActivity} t32
    snapshot=TaskSnapshot{mid=1645268282476 mTopActivityComponent=com.Testing.karthik.extractionofuserid/.MainActivity mSnapshot=android.hardware.HardwareBuffer@6448d76 (1152x2496) mColorSpace=rGB ICC61966-2,1 (ide=0, model=RGB) mOrientation=1 mRotation=0 mTaskSize=Point(1440, 3120) mContentInsets={0,0,0,0} mInputMethodWindow(63bf9d8 U InputMethod)
    mLetterboxInsets={0,0,0,0} mIsLowResolution=false mIsRealSnapshot=true mWindowingMode=1 mAppearance=0 mIsTranslucent=false mHasImeSurface=false
    mInputMethodWindow=null
    mOscuringWindow.Window@c3e4e32 U com.Testing.karthik.extractionofuserid/com.Testing.karthik.extractionofuserid.MainActivity
    mSystemBooted=true mDisplayEnabled=true
    mTransactionSequence=2152
    mDisplayFrozen=false windows=0 client=false apps=0 mRotation=0 mLastOrientation=-1
    waitingForConfig=false
    Animation settings: disabled=false window=1.0 transition=1.0 animator=1.0
yogashivamathivanan@Yogashivas-MacBook-Pro ~ %
```

Figure 4.24: Application package name and activity name

- **appActivity**: The activity in the application we want to launch, as in *Figure 4.24*. The activity is “Form Sign Up”.
- **platformName**: It is the device platform such as iOS, Android, and so on.
- **deviceName**: The name of the device. In our case, since we are using an emulator, we need to provide the name of the device we have provided.
- **Udid: Unique device Identifier**. To retrieve, go to the terminal and type **adb devices** as in *Figure 4.20* and it gives the id of the device as **emulator-5554**

Open the Appium Inspector, and copy and paste the following JSON with the appropriate value as in *Figure 4.25*. Note the Appium Server setting; it should match the settings in the Appium server.

```
{
    "appPackage": "com.Testing.karthik.extractionofuserid",
    "appActivity": "com.Testing.karthik.extractionofuserid.MainActivity",
    "platformName": "Android",
    "deviceName": "Pixel 6 Pro API 33",
    "udid": "emulator-5554"
}
```

Prerequisites: Start the Appium Server installed in earlier steps.

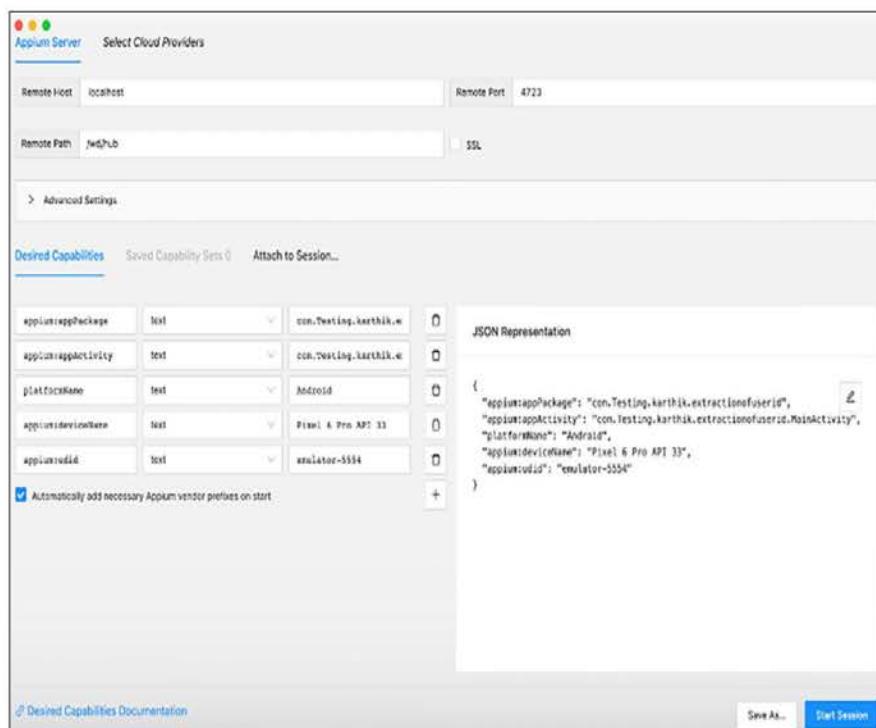


Figure 4.25: Android Inspector Desired Capabilities

Click on **Start server**, and it should open the app, as shown in Figure 4.26:

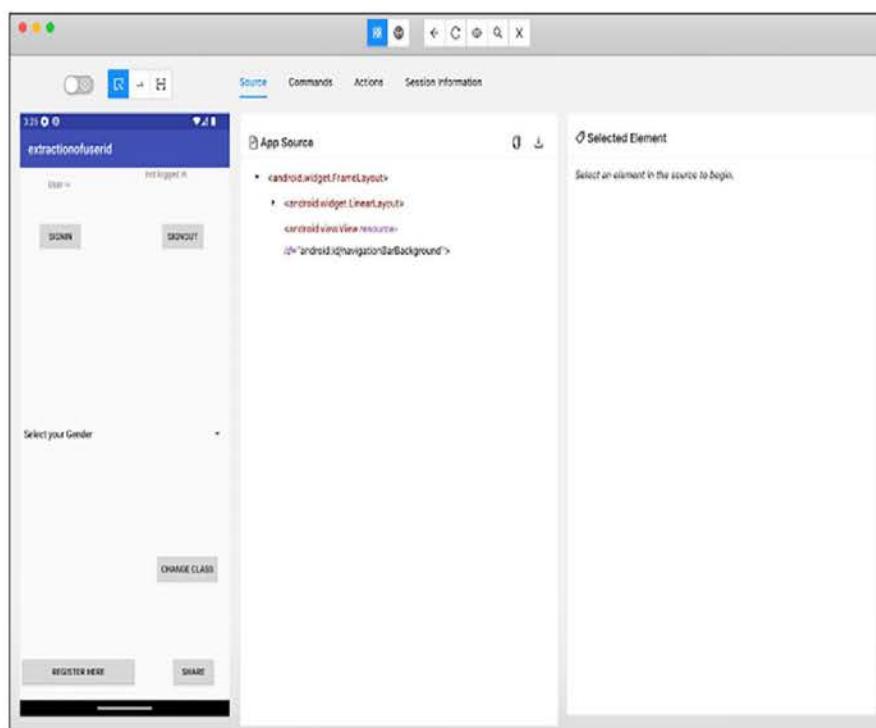


Figure 4.26: Android Inspector Session started

The Android Inspector is like the Android Emulator in Android Studio. Here, once the Inspector is started, when you click on **select Elements** on the top left, and click on any element within the application, you will see the selected Element properties, which can be used to identify the element and interact. Refer to *Figure 4.27*:

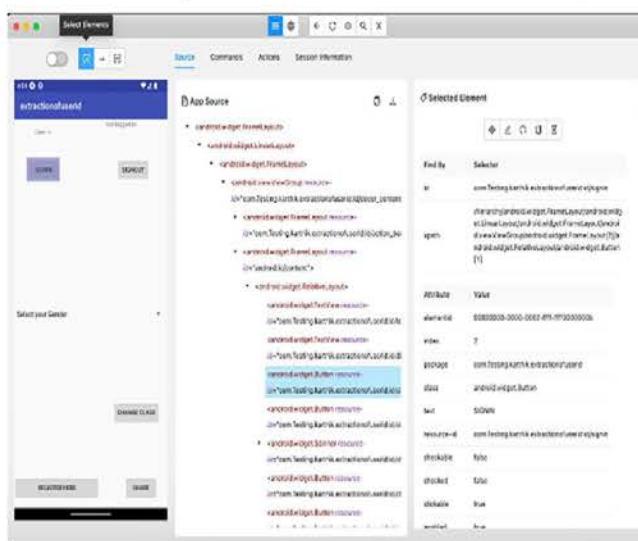


Figure 4.27: Android Inspector Interacting with Application

Additionally, we can see the logs in the Appium server. These logs can be downloaded, as shown in *Figure 4.28*:



Figure 4.28: Appium Server Raw Logs

Application Launching: Android

With all the required tools and servers configured, let us move to code and the first step to automating mobile applications using Appium. The application needs to be launched before integrating with the elements. The pre-requisite is that the Appium server needs to be running.

Code explanation

The following code is used to invoke the application. Here we need to provide the desired capabilities as discussed in the previous section. It provides test information to the Appium server. As in the following code, desired capabilities have platformName, platformVersion (Android version), deviceName (Name of the device), app (Location of the app in your machine), appPackage, and appActivity.

Following the Desired capabilities, we need to pass the Appium server location, which is localhost running in port 4723, as we configured in Appium inspector. We also need to pass the created desired capabilities to the webdriver remote instance, which requires `import from appium import webdriver`. Now the application is ready to be invoked and interact with the web elements.

In the next step, once the application is launched, click on the **Sign In** button identified using ID Attribute visible from Appium inspector as in *Figure 4.27*. Identifying element using the AppiumBy function of appium webdriver requires `import from appium.webdriver.common.appiumby import AppiumBy`. The elements can be identified by the different locators. In the following code, we have used ID and XPATH. Moreover, we have added some sleep time to handle the load time, but we will use the inbuilt waits to handle these in the next codes.

CODE

```
1. import time  
2. from appium import webdriver  
3. from appium.webdriver.common.appiumby import AppiumBy  
  
4. desired_capabilities = {'platformName': 'Android',  
5. 'platformVersion': '13', 'deviceName': 'Pixel 6 Pro API 33',  
6. 'app': '/Users/yogashivamathivanan/  
Downloads/DEMO APP.apk'}
```

```
    'appActivity': 'com.Testing.karthik.
    extractionofuserid',
7.
8.           'appActivity': 'com.Testing.karthik.
    extractionofuserid.MainActivity'}
9.
10. driver = webdriver.Remote('http://localhost:4723/wd/hub', desired_
    capabilities)
11.
12. signInButton = driver.find_element(AppiumBy.ID, 'com.Testing.karthik.extractionofuserid:id/signin')
13. signInButton.click()
14.
15. time.sleep(2)

16. countryDropdown = driver.find_element(AppiumBy.ID, 'android:id/
    text1')
17. countryDropdown.click()

18. time.sleep(2)

19. countryUK = driver.find_element(AppiumBy.XPATH,
20. '/hierarchy/android.widget.FrameLayout/android.widget.
    FrameLayout/android.
    widget.ListView/android.widget.CheckedTextView[14]')
21. countryUK.click()
22.
23. time.sleep(2)

24. email = driver.find_element(AppiumBy.ID,
25. 'com.Testing.karthik.extractionofuserid:id/email')
26. email.send_keys('vogashiva1990@gmail.com')
```

```
27. password = driver.find_element(AppiumBy.ID,
```

```
28. 'com.Testing.karthik.extractionofuserid:id/password')
29. password.send_keys('password')
30. signInButtonLogin = driver.find_element(AppiumBy.ID,
31. 'com.Testing.karthik.extractionofuserid:id/signin')
32. signInButtonLogin.click()

33. time.sleep(2)

34. loggedInText = driver.find_element(AppiumBy.XPATH,
35. '/hierarchy/android.widget.FrameLayout/android.widget.
36. LinearLayout/android.widget.FrameLayout/android.view.ViewGroup/
37. android.widget.FrameLayout[2]/android.widget.RelativeLayout/android.
38. widget.TextView[2]').text
39. if loggedInText == 'Logged in':
40.     print('Successfully Logged in')
41. else:
42.     print('Log in Unsuccessful')
```

OUTPUT

```
1. Successfully Logged in
```

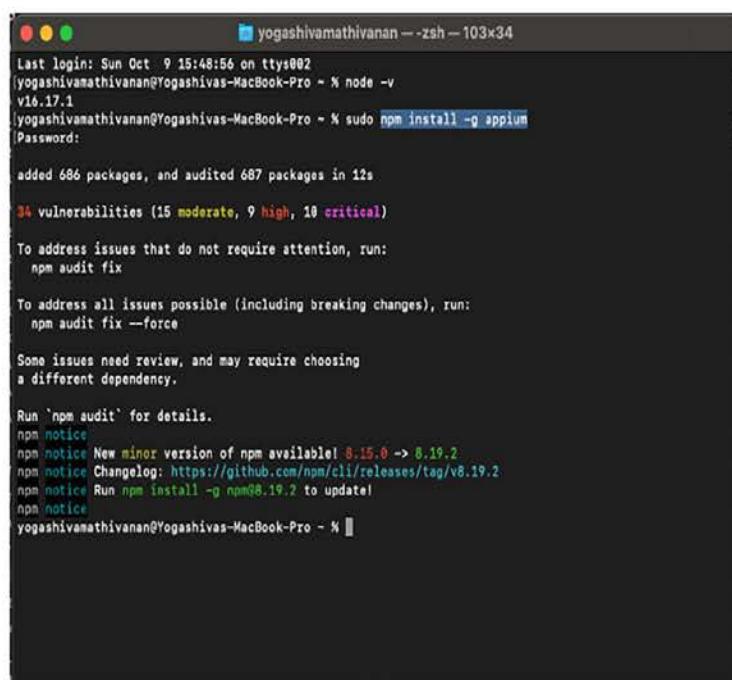
Appium Server Launch Programmatically

As noticed, when running the code, we need to have the Appium server running. This is not required and can be triggered using the Python code. Follow the given steps.

1. Install and validate the installation of node.js. Navigate to <https://nodejs.org/en/download/> and download the windows or macOS installer package and complete the installation. Open the terminal and validate using the command – “`node -v`”; it should respond with the node.js version that was

installed as in *Figure 4.29*.

2. Install Appium using the command `npm install -g Appium`, as shown in *Figure 4.29*:



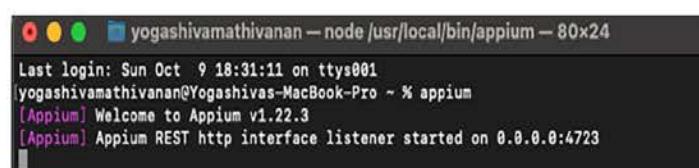
```
yogashivamathivanan --zsh-- 103x34
Last login: Sun Oct 9 15:48:56 on ttys002
[yogashivamathivanan@Yogashivas-MacBook-Pro ~ % node -v
v16.17.1
[yogashivamathivanan@Yogashivas-MacBook-Pro ~ % sudo npm install -g appium
[Password:
added 686 packages, and audited 687 packages in 12s
34 vulnerabilities (15 moderate, 9 high, 10 critical)
To address issues that do not require attention, run:
  npm audit fix
To address all issues possible (including breaking changes), run:
  npm audit fix --force
Some issues need review, and may require choosing
a different dependency.
Run 'npm audit' for details.
npm notice
npm notice New minor version of npm available! 8.15.0 → 8.19.2
npm notice Changelog: https://github.com/npm/cli/releases/tag/v8.19.2
npm notice Run npm install -g npm@8.19.2 to update!
npm notice
yogashivamathivanan@Yogashivas-MacBook-Pro ~ % ]
```

Figure 4.29: NodeJS installed and Appium Server installed

Code Explanation: In the preceding code, once the node.js is installed, we need to create instance of AppiumService and start, as in code line 8 in the following code, and stop the service after automation execution.

Appium Server launch command prompt/terminal

The Appium server can also be launched using a command prompt, by simply typing the “appium” command, as shown in *Figure 4.30*.



```
yogashivamathivanan -- node/usr/local/bin/appium -- 80x24
Last login: Sun Oct 9 18:31:11 on ttys001
[yogashivamathivanan@Yogashivas-MacBook-Pro ~ % appium
[Appium] Welcome to Appium v1.22.3
[Appium] Appium REST http interface listener started on 0.0.0.0:4723
[ ]
```



Figure 4.30: Appium Server Launch using Command Prompt

IOS Simulator Configuration on Mac

Since we have seen that XCode can be downloaded from the App store, let us see how we can use Xcode for iOS application testing. In iOS, we have a simple test application used by Appium for demo tests available on GitHub: <https://github.com/appium/ios-uicatalog>. Please note Xcode version 14 is causing some issues with the ios-uicatalog application, so downgrade the Xcode version to 13.4 and use simulator version iPhone 13 from <https://xcodereleases.com/>

1. Click on **Code** and download the zip file.
2. Unzip the file and open the file in the path from root **/ios-uicatalog-master/UICatalog/UICatalog.xcodeproj**.
3. In case the product folder is not available when opening the first path, it should open in XCode.
4. Select the simulator and click on the play button as shown in *Figure 4.31*. The app should open in the simulator.

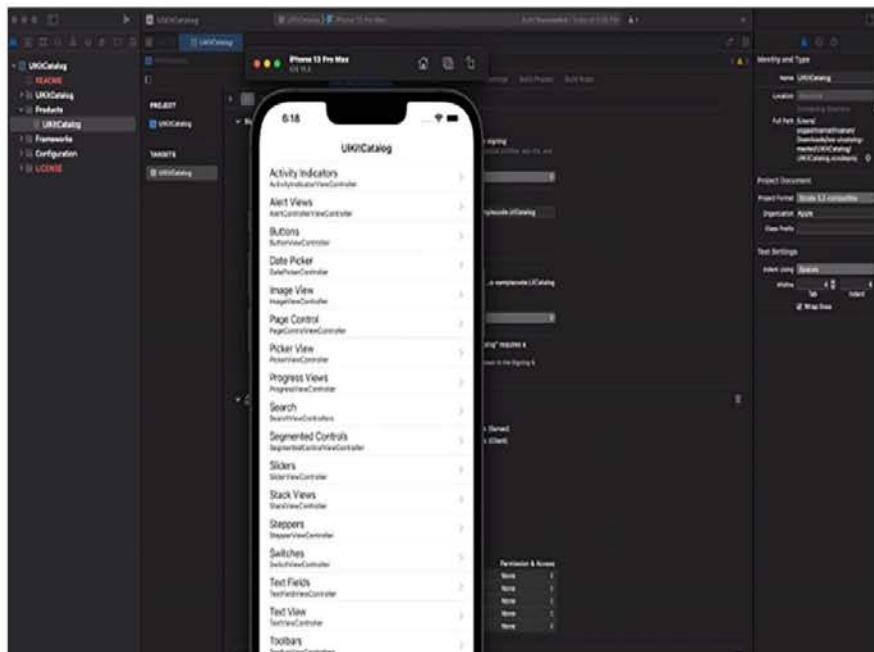




Figure 4.31: iOS Simulator

Application launching: iOS

The Simulator Application can be launched using the Python code, with the desired capabilities for iOS, as shown in the following code. The IOS application can be triggered by the inspector and interact with the application elements as in *Figure 4.32* and *Figure 4.33*.

CODE

```
1. from appium import webdriver
2. from appium.webdriver.common.appiumby import AppiumBy
3.
4. desired_caps = {'platformName': 'IOS', 'platformVersion': '15.5',
   'deviceName': 'iPhone 13 Pro Max',
5.           'automationName': 'XCUITest', 'app': (
6.             '/Users/yogashivamathivanan/Library/Developer/Xcode/
   DerivedData/UIKitCatalog-dskplzeeekmyvnbtcjosyjblrks/
   Build/Products/Debug-iphonesimulator/UIKitCatalog.app')}
7.
8. driver = webdriver.Remote("http://127.0.0.1:4723/wd/hub",
   desired_caps)
9. driver.find_element(AppiumBy.ACCESSIBILITY_ID, "Alert Views").
   click()
```

Desired Capabilities Json Representation

```
{
  "platformName": "IOS",
  "appium:platformVersion": "15.5",
```

```

    "appium:deviceName": "iPhone 13 Pro Max",
    "appium:automationName": "XCUITest",
    "appium:app": "/Users/yogashivamathivanan/Library/Developer/Xcode/DerivedData/UIKitCatalog-dskplzeeekmyvnbtcjcosyjblrks/Build/Products/Debug-iphonesimulator/UIKitCatalog.app"
}

```

Refer to *Figure 4.32* to see the IOS Inspector session that starts:

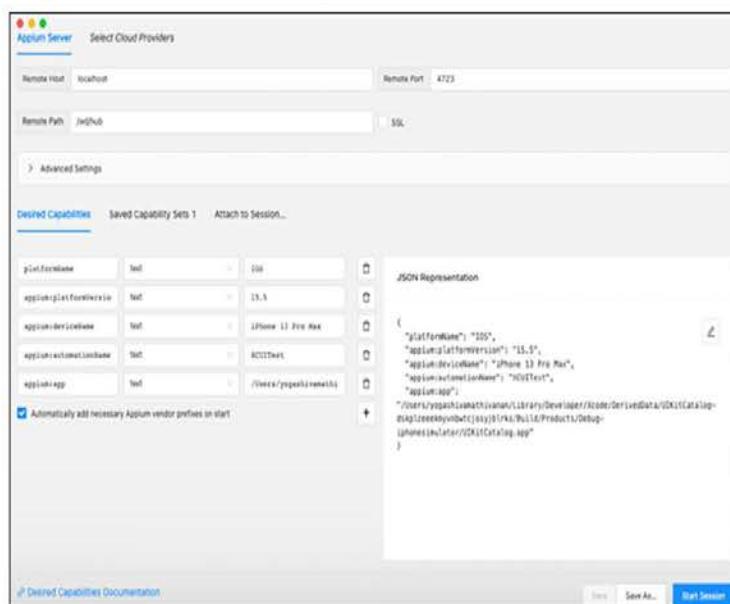


Figure 4.32: IOS Inspector Session started

Refer to *Figure 4.33* to see the IOS Inspector inspecting with application:

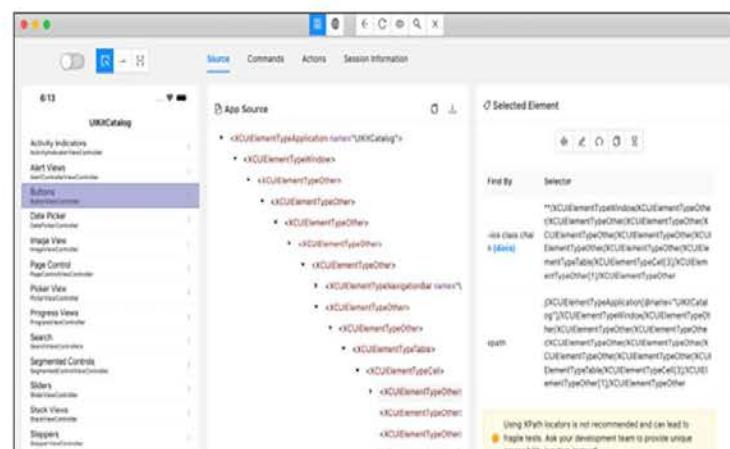




Figure 4.33: iOS Inspector Interacting with Application

Conclusion

Appium is the most popular mobile automaton solution with the capability to automate native apps, mobile web apps, and hybrid apps. Appium supports multiple programming languages that have Selenium client libraries such as Java, Python,

PHP, C#, and so on. We have seen the Appium architecture, tools, installation, and setup of Appium and how to install and trigger the mobile application from the code. The next step in Appium is to explore the locators to identify individual elements of the application and automate the interaction. In the next chapter, we will go over the locators and classes, in order to identify and interact with web elements, which will also serve as core knowledge for mobile application locators.

Key facts

- Appium is an open-source mobile application automation tool, that supports both Android and iOS for native applications, and supports Safari and chrome for mobile web apps.
- Appium uses UIAutomator to interact with Android applications and XCUITest for iOS applications.
- Appium inspector is used to identify the locators to interact with the element.
- Desired capabilities are the set of properties in key-value pair such as platformName, platformVerstion, and so on, that provide information of the application to Appium.
- Android studio allows testing and connect to Android Emulator and Xcode allows iOS simulators.

Questions

1. What are the advantages of Appium?
2. What are the differences among Native, Web and Hybrid application?
3. What is the different components of Appium architecture?
4. What is Desired Capability in Appium?

5. How to start Appium server programmatically?
6. Can Appium automate OTP based test scenarios?

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



CHAPTER 5

Locators and

Handling Web

Elements

Introduction

We have covered the Selenium WebDriver tool for the automation of web applications. Testing web application involves actions such as clicking, typing, dragging & dropping, and so on, on HTML elements. We will use Selenium WebDriver to mimic these actions as a human would, but you might wonder how. That is where Selenium Locators come into the picture. Locators are a way to identify an HTML element on the web page. Selenium supports various locators.

the web page. Selenium supports various kinds of locators.

Locators are the essential component of Selenium architecture, which helps the Selenium script in uniquely identifying web elements such as buttons, checkboxes, dropdowns, and so on. In this chapter, we will look at how to get the values of the locators and use them within the automation script.

Structure

In this chapter, we will discuss the following topics:

- Locators in Selenium
 - Web Elements in DOM (Document Object Model)

- Types of Locators in Selenium
 - Locating Element using ID
 - Locating Element using Class Name
 - Locating Element using Name Attribute
 - Locating Element using Link Text
 - Locating Element using Partial Link Text
 - Locating Element using Tag Name
 - Locating Element using CSS Selector
 - Locating Element using XPath Locator
- Chrome Locators Extensions
- Finding Multiple Web Elements and Commands
- Advanced Web Page Elements

Objectives

This chapter will help us understand the basics of finding web application elements using Selenium locators and perform actions on them. Web elements are the building blocks of a web application and locating the web elements is the fundamental step in automating web applications. Selenium provides several built-in locator strategies to uniquely identify an element. Once the web driver is initialized, the driver needs to locate the web element and trigger a JavaScript event such as click, select, and so on, or type in the text field.

Locators in Selenium

As discussed previously, Selenium locators help in uniquely identifying a web element within the web page. It is a command in the script which provides Selenium information about the type of elements such as text, text box, button, iframes, and so on, to perform actions. It might seem straightforward but the way HTML codes are written and modified as per the business requirement, finding an element uniquely becomes challenging at certain times. Thus, usage of the Selenium locator also requires a strategy to ensure that the tests are stable, faster, and reliable, with lower maintenance requirements over time.

Let us look at the problem statement with an example. There is a link that appears exactly 5 times on a given web page with the same link text, and you have developed a script with a locator to click on the third link. Now let us say, the web page is

modified and the same link text is added one more time in the second position. Now, your script is clicking the wrong link text and thus, the test will fail. So, it is extremely important to find the element uniquely.

Let us look at how to inspect the element on a web page. We are interacting with the web page in a browser, and the browser has an inbuilt inspector to locate and look at the element attributes.

Web Elements in DOM

The **Document Object Model (DOM)** is an API interface provided by the browser, that treats an XML or HTML document as a tree structure with nodes, wherein nodes have a single root node and other nodes are child nodes.

The key difference between XML and HTML is that HTML is used to display web page data with predefined tags to use, whereas XML stores and transfers data with self-defined tags as per the content. Now when a web page is loaded, the browser creates a document object model of the page. *Figure 5.1* shows the difference between XML and DOM. The significance of DOM is also illustrated in finding elements using the XPath locator. Refer to *Figure 5.1*:

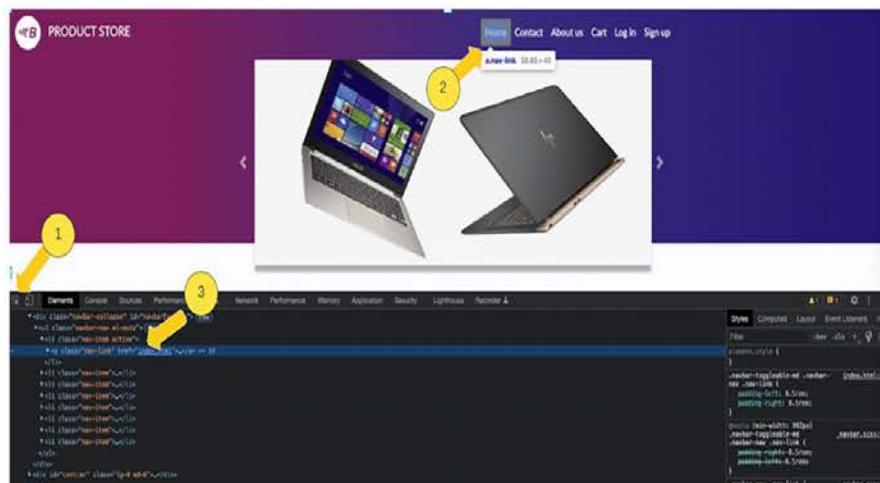




Figure 5.1: XML and DOM

Follow the given steps to find the element in DOM:

1. Navigate to the web page (<https://www.demoblaze.com/index.html>) in the browser, right-click, select **Inspect**, or command+option+I in Mac, or F12 in Windows.
2. The developer option will open with tabs such as Elements, Console, and so on. Click on the **Elements** tab. If not selected by default, and click on the **Inspector** option, as shown in *Figure 5.2*.
3. Point to any web element on the web page. Then point to the corresponding element in the Html body or any element in the Html body in the elements tab. This will highlight the corresponding element, as shown in *Figure 5.2*.



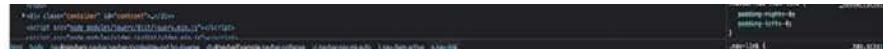


Figure 5.2: Element inspector & DOM

Every link in the HTML is referring to an element on the webpage. In the preceding example, the line `Home ` is referring to the `Home` in the navigation bar. “`a`” is the tag name, `class` & `href` are the attribute names and `nav-link` & `index.html` are the attribute values. `Home` is the text.

Types of locators in Selenium

The web elements can be uniquely located by one of the Selenium locators, as shown in *Figure 5.3*. Selenium provides a “By” class, to locate elements in DOM:

- **ClassName:** Locates elements matching the search value with the object’s class attribute.
- **ID:** Locates elements matching search value with the id attribute of the object.

- **CSS Selector:** Locates elements matching the search value with CSS selector.
- **Name:** Locates elements matching search value with the Name attribute of the object.
- **Link text:** Locates elements matching search value with the anchor tag “`a`” and the link text.
- **Partial link text:** Locates elements matching search value with the anchor tag “`a`” and partially matches the text.
- **Tag Name:** Locates elements matching the search value with the tag name.
- **XPath:** Locates elements using the XPath expression.

Refer to *Figure 5.3*:





Figure 5.3: Selenium Locators

There are browser plugins link such as chroPath, XPath finder, selectors hub, and so on, to help find the elements.

XPath and CSS Selector Locator are customized locators requiring us to write the expressions. Let us look at each of these locators in detail and see how to write code to find the element.

Locating Element using ID

Finding an element using the “id” attribute of the object is shown in the following sample HTML code. It will use ID Locator to find the login link in line 16. ID Locator is mostly reliable, safe, and less likely to be impacted by the change, and hence, is more preferred compared to other locators. If multiple elements with the same IDs is found, then the first matched ID is returned.

HTML CODE

```
1. <div class="navbar-collapse" id="navbarExample">
2.     <ul class="navbar-nav ml-auto">
3.         <li class="nav-item active">
4.             <a class="nav-link" href="index.html">Home <span
    class="sr-only">(current)</span></a>
5.         </li>
6.         <li class="nav-item">
7.             <a class="nav-link" data-toggle="modal" data-
    target="#exampleModal" href="#">Contact</a>
8.         </li>
9.         <li class="nav-item">
10.            <a class="nav-link" data-toggle="modal" data-
    target="#videoModal" href="#">About us</a>
11.        </li>
12.        <li class="nav-item">
13.            <a class="nav-link" id="cartur" href="cart.html">Cart</
```

```

        a>
14.      </li>
15.      <li class="nav-item">
16.          <a class="nav-link" href="#" id="login2" data-
              toggle="modal" data-target="#logInModal">Log in</a>
17.      </li>
18.      <li class="nav-item">
19.          <a class="nav-link" href="#" style="display:none"
              id="logout2" onClick="logOut()">Log out</a>

```

CODE

1. driver.findElement(By.ID, 'login2').click()

Locating Element using Class Name

Finding an element using the “class” attribute of the object is shown in the following sample HTML code. The class name is common across the different elements in the preceding HTML code, and using the class name attribute will be unreliable. The class name is not the preferred way to locate the element. However, if we use the

class name to locate the element and multiple elements are returned, then Selenium will do the action on the first element, searching from left to right with the class name. The class name locator is usually very useful when trying to find multiple web elements in the web page.

CODE

1. driver.findElement(By.CLASS_NAME, 'list-group')

Locating Element using Name Attribute

Finding an element using the “name” attribute of the object is shown in the following sample HTML code. The name attribute may or may not be unique for an element. The name attribute is usually added to the input text field or button. In the following HTML code, the email username input field has the name `value login[username]`, and the password input field name is `login[password]`, which is unique for the input username and password text fields.

HTML CODE

1. <label class="label" for="email">Email</label>
2. <div class="control">

```
3. <input name="login[username]" value="" autocomplete="off"
   id="email" type="email" class="input-text" title="Email" data-
   mage-init='{"mage/trim-input":{}}' data-validate="{required:true,
   'validate-email':true}">
4. </div>
5. </div>
6. <div class="field password required">
7. <label for="pass" class="label"><span>Password</span></label>
8. <div class="control">
9. <input name="login[password]" type="password" autocomplete="off"
   class="input-text" id="pass" title="Password" data-
   validate="{required:true}">
10.</div>
```

CODE

```
1. driver.find_element(By.NAME, 'login[username]').send_keys("admin")
2. driver.find_element(By.NAME, 'login[password]').send_keys('admin')
```

Locating Element using Link Text

The element with the anchor tag `<a>` is the link element. The element in the anchor tag will have a `href` attribute with a link to a different page, known as a hyperlink. The Link Text locator locates the element by exactly matching the link text available on the web page. In the following HTML code, the anchor tag text is **Forgot Your Password?** in the login page, within anchor tag `<a>` and href attribute with a hyperlink to a different page. The Link text is case-sensitive.

HTML CODE

```
1. <div class="actions-toolbar">
2. <div class="primary"><button type="submit" class="action login
   primary" name="send" id="send2"><span>Sign In</span></button></
   div>
3. <div class="secondary"><a class="action remind" href="https://
   magento.softwaretestingboard.com/customer/account/forgotpass-
   word/"><span>Forgot Your Password?</span></a></div>
4. </div>
```

CODE

```
1. driver.find_element(By.LINK_TEXT, 'Forgot Your Password?').click()
```

Locating Element using Partial Link Text

Partial link text locates the element, similar to Link text locator. However, partial link text matches the link element partial text. We can pass the substring from the text in the anchor tag in the web page. The partial link text is also case-sensitive.

CODE

```
1. driver.find_element(By.PARTIAL_LINK_TEXT, 'Pass').click()
```

Locating Element using Tag Name

The web element tags are different for different elements. If the tag name is unique for a given element, it can be used to identify the element. The tag name locator is also used to find multiple web elements in the web page. For example, if we want to count the number of links in a given page, we can locate them by using tag name `<a>`.

HTML CODE

1. <body data-container="body" data-mage-init='{"loader": { "icon": "https://magento.softwaretestingboard.com" }}' class="customer-account-forgotpassword page-layout-1column">
2. <script type="text/x-magento-init">

CODE

1. driver.find_element(By.TAG_NAME, 'body').get_attribute('class')

Locating Element using CSS Selector

CSS stands for **Cascading Style Sheets**. It is the component that makes HTML pages more attractive and organized. CSS Selector can be used to find complex elements on the web page. CSS is more flexible with locating web elements and more reliable, as it can be constructed by combining element attributes.

The CSS Selector can be found by using the browser, as shown in *Figure 5.4*:

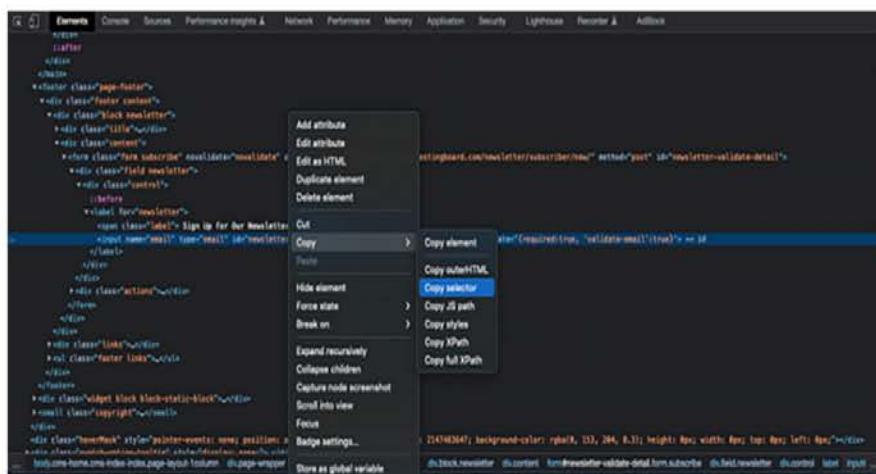


Figure 5.4: Selector from Browser

The following methods illustrate the CSS selector syntax to locate web elements.

ID

CSS Selector with the id attribute. The “#” symbol is used between tagName and ID, the tagName is optional and can be used to locate an element with “#” followed by id.

The Syntax for CSS Selector with ID is:

tagName#Id" or "#Id" or tagName[id='idValue']

HTML CODE

1. <label class="label" for="email">Email</label>
2. <div class="control">
3. <input name="login[username]" value="" autocomplete="off" id="email" type="email" class="input-text" title="Email" data-mage-init='{"mage/trim-input":{}}' data-validate="{required:true, validate-email:true}">
4. </div>

One of the following code lines can be used to locate the email username input element, using CSS selector with ID.

CODE

1. driver.find_element(By.CSS_SELECTOR, '#email').send_keys("admin")
2. driver.find_element(By.CSS_SELECTOR, 'input#email').send_keys("admin")
3. driver.find_element(By.CSS_SELECTOR, "input[id='email']").send_keys("admin")

CLASS

CSS Selector with a class attribute. The “.” symbol is used for identifying elements with CSS selector using class.

The Syntax for CSS Selector with Class is:

tagName.class" or ".className" or tagName[class='classValue']

HTML CODE

1. <input id="search" data-mage-init='{"quickSearch":{
 "formSelector":"#search_mini_form"}' type="text" class="input-text" autocomplete="off" />

One of the following code lines can be used to locate the email username input element using CSS selector with class.

CODE

1. driver.find_element(By.CSS_SELECTOR, "input[class='input-text']").send_keys("admin")
2. driver.find_element(By.CSS_SELECTOR, "input.input-text").send_keys("admin")
3. driver.find_element(By.CSS_SELECTOR, ".input-text").send_keys("admin")

ATTRIBUTE and MULTIPLE ATTRIBUTES

CSS Selector with element attributes is like using class or Id attribute, although CSS also allows the use of many other attributes as well. When two elements have similar attributes, multiple attributes can be used to identify an element uniquely. The combination can also be ID and Class, or the ID or Class used with other attributes.

The Syntax for CSS Selector with attributes is:

```
tagName[attributeName='attributeValue'].

Multiple attributes - tagName[attribute1Name = attribute1Value][
attribute2Name = attribute2Value]
```

HTML CODE

1. <label for="newsletter">
2.
3. Sign Up for Our Newsletter:
4. <input name="email" type="email" id="newsletter" placeholder="Enter your email address" data-mage-init='{"mage/trim-input":{}}' data-validate="{required:true, validate-email:true}" />
5. </label></div></div>
6. <div class="actions">
7. <button class="action subscribe primary" title="Subscribe" type="submit" aria-label="Subscribe">
8. Subscribe

CODE

1. driver.find_element(By.CSS_SELECTOR, "input[type='email']").send_keys("adminadmin@gmail.com")


```
2. driver.find_element(By.CSS_SELECTOR, "button[title='Subscribe']  
[type='submit'])
```

Relative CSS Selector

Relative CSS Selectors are used to select elements based on their relationship with other elements or attributes on the page. These selectors provide a way to select elements that match a certain pattern or attribute value. Relative CSS Selectors include attribute selectors with special characters that starts with ^, ends with \$, and contain * & ~. These selectors allow you to target elements that have specific attribute values or patterns, as given:

- ^= selects elements that have an attribute value that starts with a specific string. For example, `input[name^="username"]` selects all input elements that have a name attribute starting with the string “username”.
- ~= selects elements that have an attribute value that contains a specific word. For example, `div[class~="example"]` selects all div elements that have a class attribute containing the word “example”.
- \$= selects elements that have an attribute value that ends with a specific string. For example, `a[href$=".pdf"]` selects all elements that have an href attribute ending with the string “.pdf”.
- *= selects elements that have an attribute value that contains a specific substring. For example, `img[alt*="cat"]` selects all img elements that have an alt attribute containing the substring “cat”.

Locating Element using XPath Locator

XPath is defined as an XML path, used in XML path expression to locate nodes that navigate via elements and attributes in HTML DOM structure. XPath is an address of the element. This is very useful, especially when an element has minimum or no attributes assigned to it, as it becomes very difficult to locate such elements otherwise.

Let us discuss the structure of HTML. As discussed, there is one root node, and the other nodes are called child nodes. As shown previously in *Figure 5.1*, XML there has one root node “menu”; the following nodes “starter”, “main_course”, “drinks” and “dessert” are child to root node and these child nodes have further child nodes of their own. This kind of HTML document in the form of a tree structure is DOM. In the general tree structure of In HTML, the root node is `<html>` with two child nodes `<head>` and `<body>`. `<head>` had only one child element `<title>`. This denotes `<head>` is the child to `<html>` and parent to `<title>`. `<head>` and `<body>` tags are

siblings. The body will have a number of nodes that can have siblings, parent, or child relationships among them.

XPath can be of two types to locate web elements, Absolute XPath and Relative XPath.

Absolute XPath

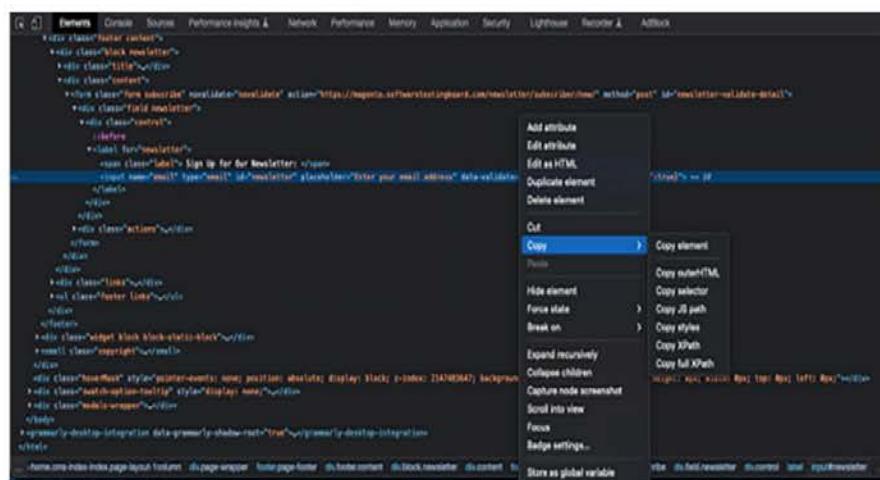
An absolute XPath is a method in which elements are located navigating from the root node, <html> in a hierarchical order. The connection between nodes is indicated by a “/”. It can be retrieved in the browser inspect element using **Copy full XPath**, as shown in *Figure 5.5* and following is an example of absolute XPath. This is not reliable as it is more susceptible to failure even if a minor change occurs on the web page.

For example:

```
/html/body/div[2]/footer/div/div[1]/div[2]/form/div[1]/div/label/input.
```

Relative XPath

The web elements are located by referring to the elements that need to be identified and not from the root node. This makes relative XPath less susceptible to failure, in case of changes. This can be identified from the browser using **Copy XPath** as shown in *Figure 5.5*, but this is not always reliable.




```
/*[@attributeName = 'attributeValue']  
// defines the current directory or tag in HTML. tagName is the current directory  
tag, the element's attribute to be selected is specified by "@" and the value of the  
attribute. The nodes are separated by //.
```

XPath with logical operators and XPath Functions

The preceding syntax can be used with logical operators AND and OR, as well as with XPath functions, to make the relative XPath expression more robust.

- **AND** logical operator is used to locate an element with one or more XPath with specified conditions match in the given web page. Otherwise, it returns no element. The keyword here is “**and**”.

Syntax:

```
//tagName[@attributeName1='attributeValue1' and @attributeName2 =  
'attributeValue2' and ...]
```

- **OR** logical operator is used to locate an element when one of the multiple XPath with specified conditions matches the given web page. Otherwise, it returns no element. The keyword here is “**or**”.

Syntax:

```
//tagName[@attributeName1='attributeValue1' or @attributeName2 =  
'attributeValue2' or ...]
```

- **text()** function is used to locate elements in the web page using the element text. This can be beneficial when dealing with an element with no attribute but text.

Syntax:

```
//tagName[text() = 'elementText' ]
```

- **contains()** function is used to locate the element in the web page that contains attribute value or element text partially matching the specified text value within the “**contains**” keyword.

Syntax:

```
//tagName[contains(@attributeName1,'partialAttributeValue1' ]  
//tagName[contains(text(),'partialElementText' ]
```

- **starts-with()** function is like **contains()** function and is used to locate the element that has an attribute value that starts with the specified text value in the XPath.

Syntax:

```
//tagName[starts-with(@attributeName1,'partialStartStringOfAttributeValue1' ]
```

HTML CODE

1. <input name="login[username]" value="" autocomplete="off" id="email" type="email" class="input-text" title="Email" data-validate="{required:true, 'validate-email':true}" aria-required="true" xpath="1">
2. <input name="login[password]" type="password" autocomplete="off" class="input-text" id="pass" title="Password" data-validate="{required:true}" aria-required="true" xpath="1">
3. <button type="submit" class="action login primary" name="send" id="send2" xpath="1">Sign In</button>
4. <input name="email" type="email" id="newsletter" placeholder="Enter your email address" data-validate="{required:true, 'validate-email':true}" xpath="1">
5. <button class="action subscribe primary" title="Subscribe" type="submit" aria-label="Subscribe" xpath="1">
6. Subscribe
7. </button>
8. Tops

CODE

1. from selenium import webdriver
2. from selenium.webdriver.chrome.service import Service
3. from selenium.webdriver.common.by import By
4. from webdriver_manager.chrome import ChromeDriverManager
- 5.
6. driver = webdriver.Chrome(service=Service(ChromeDriverManager("").install()))
7. driver.get("https://magento.softwaretestingboard.com/customer/account/login")
8. # driver.find_element(By.XPATH, "/html/body/div[3]/main/div[3]/div/div[2]/div[1]/div[2]/form/fieldset/div[2]/div/input").send_keys("admin12@gmail.com")

```

9. driver.find_element(By.XPATH, "//*[@id='email' and @type='email']").send_keys("admin12@gmail.com")
10. driver.find_element(By.XPATH, "//input[@id='pass' or @title='password']").send_keys("admin12!")
11. driver.find_element(By.XPATH, "//fieldset[@class='fieldset login']//span[contains(text(),'Sign In')]").click()
12. driver.find_element(By.XPATH, "//span[text()='Men']").click()
13. driver.find_element(By.XPATH, "//input[@type='email']").send_keys("adminadmin@gmail.com")
14. # driver.find_element(By.XPATH, "//*[contains(@aria-label,'Subs')]").click()
15. driver.find_element(By.XPATH, "//*[contains(text(),'Subs')]").click()
16. driver.find_element(By.XPATH, "//a[text()='Tops']").click()

```

XPath Axes

When multiple elements have the same properties, identifying an element uniquely becomes difficult. An Axes is the path to access the current node context, to locate a different node in the web page DOM. These Axes locate an element which are not possible using Xpath methods such as ID, ClassName, and others, with respect to its relationship with other nodes such as a child, or sibling.

Axes are named because they inform about the axis on which elements are lying relative to an element. *Table 5.1* features the various commonly used XPath axes methods in Selenium WebDriver.

Axis Name	Description	Syntax
Ancestor	All ancestors of the current node.	//*[attribute='value']//ancestor::tagName
Child	All child node of current node.	//*[attribute='value']//child::tagName
Descendant	All descendants of current node.	//*[attribute='value']//descendant::tagName
Following	All nodes after closing tag of the current node excluding current node descendants.	//*[attribute='value']//following::tagName
Following-Sibling	All siblings after of the current node.	//*[attribute='value']//following-sibling::tagName

Axis Name	Description	Syntax
Parent	The parent of current node.	<code>//*[@@attribute='value']//parent::tagName</code>
Preceding	All nodes appear before the current node, excluding ancestors, attribute nodes, and namespace nodes.	<code>//*[@@attribute='value']//preceding::tagName</code>
Preceding-Sibling	All siblings before the current node.	<code>//*[@@attribute='value']//preceding-sibling::tagName</code>

Table 5.1: Commonly used XPath axes methods in selenium webdriver

Refer to the following code:

HTML CODE

```

1. <div class="header_user_info"><a class="login" href="http://
automationpractice.com/index.php?controller=my-account"
rel="nofollow" title="Log in to your customer account">Sign in</
a></div>

2. <div id="contact-link"><a href="http://automationpractice.com/
index.php?controller=contact" title="Contact Us">Contact us</a></
div>

3. <label for="email">Email address</label>

4. <input class="is_required validate account_input form-control"
data-validate="isEmail" type="text" id="email" name="email"
value="" /></div>

5. <div class="form-group">
6. <label for="passwd">Password</label>
7. <span><input class="is_required validate account_input form-
control" type="password" data-validate="isPasswd" id="passwd"
name="passwd" value="" /></span></div>
8. <p class="lost_password form-group"><a href="http://automation-
practice.com/index.php?controller=password" title="Recover your
forgotten password" rel="nofollow">Forgot your password?</a></p>
9. <p class="submit">
10.<input type="hidden" class="hidden" name="back" value="my-account"
/>
11.<button type="submit" id="SubmitLogin" name="SubmitLogin">
```

```
class="button btn btn-default button-medium">
```

```
12.<span>  
13.<i class="icon-lock left"></i>Sign in</span></button>
```

CODE

```
1. from selenium import webdriver  
2. from selenium.webdriver.chrome.service import Service  
3. from selenium.webdriver.common.by import By  
4. from webdriver_manager.chrome import ChromeDriverManager  
5. driver = webdriver.Chrome(service=Service(ChromeDriverManager("").  
install()))  
6. driver.get("http://automationpractice.com/index.php")  
7. driver.find_element(By.XPATH, "//*[@id='contact-link']//preceding-  
 sibling::div/a").click()  
8. driver.find_element(By.XPATH, "//*[ @for='email']//following-sib-  
ling::input").send_keys("admin12@gmail.com")  
9. driver.find_element(By.XPATH, "//label[text()='Password']//  
parent::div/span/input").send_keys("admin12!@")  
10. driver.find_element(By.XPATH, "//p[@class='submit']//child::but-  
ton").click()
```

Validating XPath and CSS in Browser Console

When creating XPath and CSS looking at the attributes and axes, we need a way to validate the XPath and CSS Selector within the browser. Once the XPath or CSS is constructed, type the following command in the console. It will then match the respective element or elements, as shown in *Figure 5.6*:

```
XPath-$x("XPath Value")  
CSS - $$("Selector Value")
```

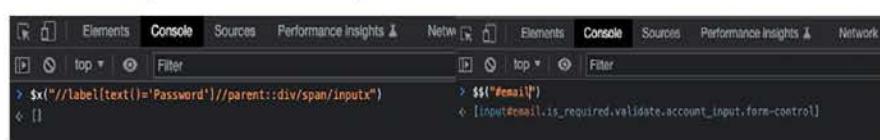


Figure 5.6: XPath and Selector validation in the browser

Chrome Locators Extensions

Finding elements in the web page is the core of automation. There will be instances when we can use a tool to support finding XPath and Selector in the web page. One of the most common Chrome extensions for finding the XPath of web elements, is Selectors Hub. Similarly, there is Chropath, RexPath, XPath finder, and so on.

As shown in *Figure 5.7*, SelectorsHub can be added from the Chrome web store, and the web element required path can be copied by right-clicking on the element and on the right panel on inspect element.

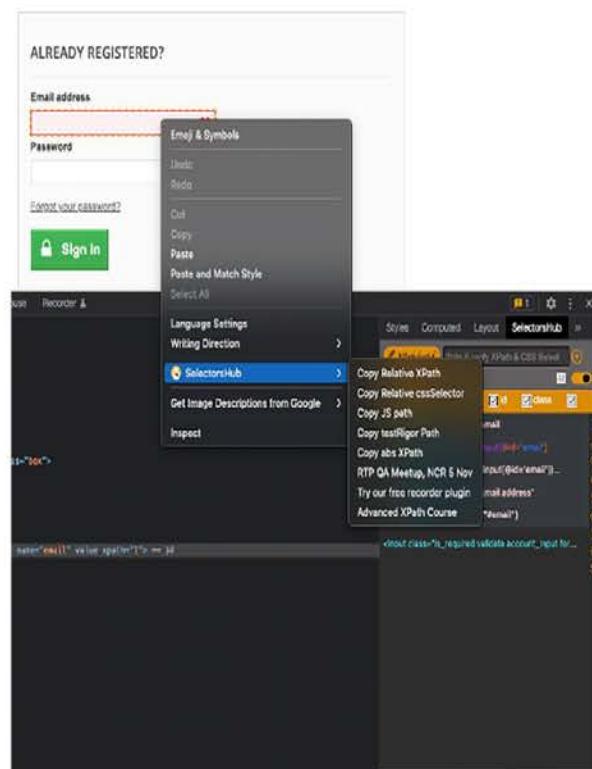


Figure 5.7: SelectorsHub Add and Use

Finding multiple web elements and commands

The Web Elements in a Web Page can be located by the Selenium locator using the **find_element** function. To find multiple elements in the same web page, **find_elements** function needs to be used instead. For example, if we need to collect the number of links present on a given web page, one element can be located using the **tagName** locator `<a>` anchor tag which will have hyperlinks. To find all the elements with `<a>` anchor tag, the **find_elements** function needs to be used. The **find_elements** returns variable of type list. Moreover, if no element is found while using the **find_element** function, there will be no element found. Exception may be thrown, but with the **find_elements** function, there will not be any error and instead the list length will be zero.

CODE

```
1. countLinks = driver.find_elements(By.TAG_NAME, "a")
2. priceContentNumber = driver.find_elements(By.CLASS_NAME,'content_price')
3. print("Number of Links =" + str(len(countLinks)) +" and Number of
priceContent " + str(len(priceContentNumber)))
```

Commands

We can use additional Selenium commands to check element status before performing actions and achieving browser objectives respectively. We have seen commands such as maximize, getting page title, quit and so on, and we will see some additional such commands.

Following are some of the additional commands added to the code.

- **driver.get_cookies()**: Returns list of cookies in the page.
- **element.get_attribute('attributeName')**: Returns the attribute of the located element.
- **driver.back()**: Navigates the page to the previous page.

- `driver.current_url`: Returns current page URL.
- `driver.page_source`: Returns the current page source content.

CODE

```
1. from selenium import webdriver
2. from selenium.webdriver.chrome.service import Service
3. from selenium.webdriver.common.by import By
4. from webdriver_manager.chrome import ChromeDriverManager
5.
6. driver = webdriver.Chrome(service=Service(ChromeDriverManager("").install()))
7. driver.maximize_window()
8. driver.get("http://automationpractice.com/index.php")
9. print(driver.page_source)
10. driver.find_element(By.XPATH, "//*[@id='contact-link']//preceding-
    sibling::div/a").click()
11. print(driver.get_cookies())
12. print(driver.get_cookie('name'))
13. driver.find_element(By.XPATH, "//*[@for='email']//following-
    sibling::input").send_keys("admin12@gmail.com")
14. driver.find_element(By.XPATH, "//label[text()='Password']//"
    parent::div/span/input").send_keys("admin12!@")
15. print(driver.find_element(By.XPATH, "//p[@class='submit']//"
    child::button").get_attribute('class'))
16. driver.find_element(By.XPATH, "//p[@class='submit']//"
    child::button").click()
17. print(driver.title)
18. driver.find_element(By.CSS_SELECTOR, "a[title='Women']").click()
19. print(driver.find_element(By.CSS_SELECTOR, "a[title='Women']").
    text)
20. driver.back()
21. print(driver.current_url)
```

```
22. driver.quit()
```

Advanced web page elements

Let us now explore some advanced Web Page elements.

Checkbox

Checkboxes are very common in modern web pages to select options from multiple choices. In HTML usually, the class name or type attribute denotes the element as a checkbox. Some web pages have attributes to denote whether the checkbox is “checked” or “unchecked” and condition can be added before clicking on the element.

The checkbox can be identified using the Selenium locator and clicked as in the first approach in the following code. We can also collect all the checkboxes in the web page using the `find_elements` function and add a condition to click the desired checkbox. We can also check if the checkbox is selected using `is_selected()` function, as in the second approach in the following code. In the second approach, we can also run the loop using the index and check the checkboxes in the desired index.

CODE

```
1. driver.find_element(By.XPATH, "//*[text()='Desktop']/preceding-
   sibling::span[2]").click()
2. driver.find_element(By.XPATH, "//*[text()='Downloads']/preceding-
   sibling::span[2]").click()
3. # 2nd Approach
4. checkboxes = driver.find_elements(By.XPATH, "//span[@class='rct-
   checkbox']")
5. for checkbox in checkboxes:
6.     if checkbox.get_attribute('id') == 'Desktop' and checkbox.
       get_attribute('id') == 'Downloads':
7.         if not checkbox.is_selected():
8.             checkbox.click()
```

Radio button

Radio buttons allow the user to select only one of the given options. Unlike checkboxes, since only one option can be selected, each choice must be its own item and label. Once the user selects radio buttons in a group, other options require to be unselected. However, it is difficult to clear the selection in a group. Rather, once

an option is selected, we can switch between the options, but at least one option is required to be selected. Radio buttons usually have the type attribute “radio” and the id is unique. Radio buttons can be located and selected using selenium locators.

Dropdown

The dropdown is one of the most common elements on the web page, allowing the user to select one option from the dropdown. In HTML, dropdowns are implemented using the Select tag or input tag. Performing actions on the dropdown requires the import of the Select class from the webdriver package, and passing the dropdown locator as an argument, as in the following code. Once imported, **Select class** allows selecting one of the options by Index (specifying the position), by value attribute, or by visible text along with other methods, as shown in *Figure 5.8*.

CODE

```
1. from selenium import webdriver
2. from selenium.webdriver.chrome.service import Service
3. from selenium.webdriver.common.by import By
4. from selenium.webdriver.support.select import Select
5. from webdriver_manager.chrome import ChromeDriverManager
6.
7. driver = webdriver.Chrome(service=Service(ChromeDriverManager("").install()))
8. driver.get("https://www.globalsqa.com/demo-site/select-dropdown-menu/")
9. dropdownSelect = Select(driver.find_element(By.XPATH, "//div[@class='single_tab_div resp-tab-content resp-tab-content-active']//p//select"))
10. dropdownSelect.select_by_value("USA")
```



⑨ select_by_value(self, value)	Select
⑩ select_by_visible_text(self, text)	Select
⑪ __getattribute__(self, name)	object
⑫ all_selected_options	Select
⑬ deselect_all(self)	Select
⑭ deselect_by_index(self, index)	Select
⑮ deselect_by_value(self, value)	Select
⑯ deselect_by_visible_text(self, text)	Select
⑰ first_selected_option	Select
⑱ is_multiple	Select
⑲ options	Select

Figure 5.8: Dropdown Actions

IFrame

The Iframes are HTML elements that allow rendering a document within another HTML document on a webpage, that is, any HTML content hosted from external content within the web page. HTML has `<iframe>` tag for such external content. One web page can have multiple `<iframe>` elements. To interact with an element within an iframe, we would need to switch to the iframe and after the interaction, switch back to the main content.

The Iframes can be identified by using the iframe “ID”, “Name”, “Index”, “IFrame Web Element”. In the following code, we are counting the number of links on the web page and the number of links on the iframe. The number of links in the main web page does not include links within the iframe. In this following code, the iframe element is identified using XPath and switched to iframe using `driver.switch_to.frame` (ID/Name/Index/WebElement), after the action in iframe. To perform the next action in the web page, the control needs to switch back to main web page content using `driver.switch_to.default_content()`. When control is in the iframe and tries to find an element outside of the iframe, Selenium will not be able to locate the element.

CODE

```

1. from selenium import webdriver
2. from selenium.webdriver.chrome.service import Service
3. from selenium.webdriver.common.by import By
4. from webdriver_manager.chrome import ChromeDriverManager
5.
6. driver = webdriver.Chrome(service=Service(ChromeDriverManager("").install()))
7. driver.get("https://www.globalsqa.com/demo-site/frames-and-
```

```
windows/#iFrame")
8. NumberOfLinksMain = driver.find_elements(By.TAG_NAME, "a")
9. print(len(NumberOfLinksMain))
10. iframeElement = driver.find_element(By.XPATH, "//*[@id='post-2632']/
    div[2]/div/div/div[3]/p/iframe")
11. # driver.switch_to.frame("globalSqa")
12. # driver.switch_to.frame("globalID")
13. driver.switch_to.frame(1)
```

```
14. # driver.switch_to.frame(iframeElement)
15. NumberOfLinksIframe = driver.find_elements(By.TAG_NAME, "a")
16. print(len(NumberOfLinksIframe))
17. driver.switch_to.default_content()
18. driver.quit()
```

Alert popup

The Alert popup is not an element part of the DOM, and hence cannot be identified using the Selenium locator. It is used to take or provide confirmation of action or take input before confirmation. Alert popup usually contains Accepts, Decline, message, or input fields. To interact with the Alert popup, driver control is required to switch to the popup and perform actions using the methods `text` to get alert text, `accept()`, `dismiss()`, `send_keys("value")`, and so on.

CODE

```
1. from selenium import webdriver
2. from selenium.webdriver.chrome.service import Service
3. from selenium.webdriver.common.by import By
4. from webdriver_manager.chrome import ChromeDriverManager
5.
6. driver = webdriver.Chrome(service=Service(ChromeDriverManager("").install()))
7. driver.get("https://chercher.tech/practice/practice-pop-ups-
    selenium-webdriver")
8. driver.find_element(By.XPATH, "//input[@name='alert']").click()
9. print(driver.switch_to.alert.text)
```

```

10. driver.switch_to.alert.accept()
11. driver.find_element(By.XPATH, "//input[@name='confirmation']").click()
12. print(driver.switch_to.alert.text)
13. driver.switch_to.alert.accept()
14. driver.find_element(By.XPATH, "//input[@name='confirmation']").click()
15. print(driver.switch_to.alert.text)
16. driver.switch_to.alert.dismiss()

```

```

17. driver.find_element(By.XPATH, "//input[@name='prompt']").click()
18. print(driver.switch_to.alert.text)
19. driver.switch_to.alert.send_keys('Test')
20. driver.switch_to.alert.accept()
21. driver.quit()

```

Calendar date picker

We can have a requirement to input the date in the date text box, but on modern web pages, we cannot send the date value. Instead, select it from the date picker. This requires finding the Month, Day, and Year attribute and matching it with the required date to be selected. In the following code, date and year in the date picker are identified and clicked on back button until the condition is matched, and then broken out of the loop.

CODE

```

1. driver = webdriver.Chrome(service=Service(ChromeDriverManager("").install()))
2. driver.get("https://jqueryui.com-datepicker/")
3. driver.switch_to.frame(driver.find_element(By.XPATH, "//iframe"))
4. driver.find_element(By.ID, 'datepicker').click()
5. while True:
6.     month = driver.find_element(By.XPATH, "//*[@class='ui-datepicker-month']").text
7.     year = driver.find_element(By.XPATH, "//*[@class='ui-datepicker-year']").text

```

```

8.     print(month)
9.     print(year)
10.    if month == "April" and year == "2019":
11.        driver.find_element(By.XPATH, "//*[@class='ui-datepicker-
12.        calendar']/tbody//a[text()='20']").click()
13.        driver.find_element(By.XPATH, "//span[@class='ui-icon ui-icon-
14.        circle-triangle-w']").click()
14.    driver.quit()

```

Window Handles

There are instances when clicking a hyperlink in the web page navigates to another tab or window, and actions need to be performed in the new window and return to the main page for the next set of actions. These new windows or tabs opened by the hyperlink have their own handle ID assigned by the Selenium WebDriver, which allows us to switch between these tabs and windows. So, we need to have a way to store these handle IDs and navigate through tabs based on requirements. As in the following code, we can navigate by capturing the current window handle and all window handles using `driver.current_window_handle` and `driver.window_handles`, respectively, switching between them using the index. In case we are dealing with multiple windows handles, we can loop over the handles and perform the action on the intended handle ID.

CODE

```

1. from selenium import webdriver
2. from selenium.webdriver.chrome.service import Service
3. from selenium.webdriver.common.by import By
4. from webdriver_manager.chrome import ChromeDriverManager
5.
6. driver = webdriver.Chrome(service=Service(ChromeDriverManager("").install()))
7. driver.get("https://www.globalsqa.com/")
8. driver.find_element(By.XPATH, "//li[@id='menu-item-6898']//a[normalize-space()='CheatSheets']").click()
9. mainHandle = driver.current_window_handle

```

```

7. mainHandle = driver.current_window_handle
10. driver.find_element(By.XPATH, "//a[text()='GIT Cheat Sheet']").click()
11. windowIds = driver.window_handles
12. driver.switch_to.window(windowIds[1])
13. pageText = driver.find_element(By.XPATH, "//strong[text()='Download GIT CheatSheet PDF']").text
14. driver.switch_to.window(mainHandle)
15. driver.find_element(By.XPATH, "//a[normalize-space()='VIM Cheat Sheet']").click()
16. driver.quit()

```

Loop

```

17. for id in windowIds:
18.     if mainHandle != id:
19.         driver.switch_to.window(id)

```

Web tables

Web Tables are unique on a web page and in an HTML document, it is defined with the tag **<table>**, followed by **<tbody>** tag within which, the rows are defined with tab **<tr>**, **<th>** for header, and **<td>** for the column as in the following HTML code.

There are two types of web table, Static and Dynamic. As the name suggests, static web tables are tables with static data and the number of rows and columns are static, whereas dynamic tables have changed data depending on certain conditions, on page reloading, and changed rows after specific user actions. Selenium has no specific command for handling dynamic web tables, and the table structure in HTML is the same for both the web tables. However, it will require automation logic to handle the condition that makes the web table dynamic.

In the following code, to work with data in the web table, we need to find the number of rows and number of columns and loop through rows and inner loop for column data. We can then apply conditions to retrieve or perform the action on the table data.

HTML CODE

```

1. <table id="customers" style="border-collapse: collapse; border: 1px solid black; width: 100%;">

```

```

        sizing: inherit; color: black; font-family: arial, sans-serif;
        font-size: 15px; width: 99%;>
2. <tbody style="box-sizing: inherit;">
3.   <tr style="box-sizing: inherit;">
4.     <th style="border: 1px solid rgb(221, 221, 221); box-sizing:
        inherit; padding: 8px; text-align: left;">Company</th>
5.     <th style="border: 1px solid rgb(221, 221, 221); box-sizing:
        inherit; padding: 8px; text-align: left;">Contact</th>
6.     <th style="border: 1px solid rgb(221, 221, 221); box-sizing:
        inherit; padding: 8px; text-align: left;">Country</th>
7.   </tr>
8.   <tr style="background-color: #dddddd; box-sizing: inherit;">

```

```

9.     <td style="border: 1px solid rgb(221, 221, 221); box-sizing:
        inherit; padding: 8px;">Google</td>
10.    <td style="border: 1px solid rgb(221, 221, 221); box-sizing:
        inherit; padding: 8px;">Maria Anders</td>
11.    <td style="border: 1px solid rgb(221, 221, 221); box-sizing:
        inherit; padding: 8px;">Germany</td>
12.  </tr>
13. </tbody>
14.</table>

```

CODE

```

1. driver.get("https://www.techlistic.com/p/demo-selenium-practice.
html")
2. numberOfRows = driver.find_elements(By.XPATH, "//table[@
id='customers']/tbody/tr")
3. numberOfCol = driver.find_elements(By.XPATH, "//table[@
id='customers']/tbody/tr[1]/th")
4. print(str(len(numberOfRows)) + " and " + str(len(numberOfCol)))
5. for rowData in range(2, len(numberOfRows) + 1):
6.     for colData in range(1, len(numberOfCol)+1):
7.         rowColData = driver.find_element(By.XPATH, "//
table[@id='customers']/tbody/tr["+str(rowData)+"]/

```

```

        ta[+str(colData)+"]")
8.     if rowColData.text == 'Amazon':
9.         print(driver.find_element(By.XPATH, "//table[@
   id='customers']/tbody/tr["+str(rowData)+"]/td[2]").text)
10.    print(driver.find_element(By.XPATH, "//table[@
      id='customers']/tbody/tr["+ str(rowData) + "]/
      td[3]").text)
11. driver.quit()

```

Action Chains for Advanced Operations

Action chains are the methods used by Selenium, that enables automation of low-level mouse and key press interactions. This helps in actions such as mouse movements, hover, right-click, double-click, click & hold, dragging & dropping, and so on. Action

chains help in complicated web actions. Action chains are implemented by serially adding all the actions in the chain pattern in the same line followed by `perform()` method. This requires import of `from selenium.webdriver.common.action_chains import ActionChains`, initialized by passing the driver instance as an argument. To perform keyboard action, import from `selenium.webdriver import ActionChains, Keys` is required.

CODE

```

1. actionPerform = ActionChains(driver)
2. #Mouse Hover
3. actionPerform.move_to_element(driver.find_element(By.XPATH,
   "//a[text()='Main Item 2']")).move_to_element(driver.find_
   element(By.XPATH, "//a[text()='SUB SUB LIST »']")).move_to_
   element(driver.find_element(By.XPATH, "//a[text()='Sub Sub Item
   1'])).perform()
4. #Mouse Right Click
5. actionPerform.context_click(driver.find_element(By.XPATH,
   "//a[text()='Main Item 1'])).perform()
6. #Mouse Double Click
7. actionPerform.double_click(driver.find_element(driver.find_
   element(By.XPATH, "//span[@id='double_click']))).perform()
8. #Mouse Drag and Drop

```

```

8. #Mouse drag and drop
9. sourceElement = driver.find_element(By.XPATH, "//h5[text()='High
Tatras 3']/following-sibling::img")
10. targetElement = driver.find_element(By.XPATH, "//div[@id='trash']")
11. actionPerform.drag_and_drop(sourceElement, targetElement).
perform()
12. #keyboard ALT+SHIFT+ENTER
13. actionPerform.key_down(Keys.ALT).key_down(Keys.SHIFT).key_
down(Keys.ENTER).perform()

```

Conclusion

Finding and performing actions on web elements is the core of automation using Selenium. Working and understanding the HTML and DOM of the web page is essential. Knowing how to use the locators correctly is key and a building block of

stable automation scripts. If the test is unable to locate the element uniquely or is not stable enough to withstand any changes or modifications to the web page, then automation loses its significance. We have seen the page structure, how to look at the element structure in HTML, and how to locate elements and perform operations on these located elements. We have also seen the working with these web elements, with the programming language Python, that makes the code cleaner and simple.

In the next chapter, we will go over similar ways to locate elements in mobile applications in Android and iOS. In Selenium next, we will go over how to structure these test case flow using frameworks and withstand possible failures.

Key facts

- Selenium locators is a command in the script which tells Selenium information about the type of elements such as text, text box, button, iframes, among others, to uniquely identify web elements and perform actions.
- The **Document Object Model (DOM)** is an API interface provided by the browser, that treats an XML or HTML document as a tree structure with node.
- Different locators are ID, ClassName, CSS Selector, Name, Link Text, Partial Link Text, TagName, and XPath.
- XPath and CSS Selector Locator are customized locators requiring us to write the appropriate

use expressions.

- `find_element` is used to locate a single element whereas `find_elements` is used to locate multiple elements with similar attributes on a web page.
- An absolute XPath is a method in which elements are located navigating from the root node, `<html>` in a hierarchical order. In Relative XPath, web elements are located by referring to the elements that need to be identified.
- **XPath and CSS Selector can be verified using the command XPath Value for XPath-\$x, and \$\$("Selector Value") for CSS, in the console.**
- Window handles are used to switch between tabs or windows during execution.
- IFrame is an HTML content hosted from external content within the web page.
- Action Chain is used to perform mouse and keyboard actions on the element.

Questions

1. What are Selenium Locators?
2. What are the different types of Selenium Locators?
3. What is the difference between HTML and DOM?
4. How to find elements using XPath and CSS? What is the Syntax?
5. What is the difference between Absolute and Relative XPath?
6. What are the different ways to find an element using XPath Axes?
7. How to handle multiple tabs using selenium?

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



CHAPTER 6

Appium: Locators and Gestures

Introduction

We have covered the Appium tool for the automation of mobile native apps, web and hybrid applications, its significance, installations, and setup of tools, how to invoke the application and perform basic actions such as typing texts and clicking the elements of the mobile applications. Like web applications, Mobile applications require identifying the elements uniquely and mimicking the mobile-specific user actions such as tap and double-tap gestures, scroll gestures, swipe across the screen, drag and drop, long press and release gestures, and so on. Since Appium extends Selenium WebDriver functionality, Appium can use similar types of locators to find mobile elements. An element locator is a way to identify UI elements. Finding a way to uniquely identify the elements on the mobile app, is called a locator strategy. Automating the mobile application requires understanding the use of the locators with mobile elements, since finding elements is the first step of the automation script after invoking the application. Learning which types of locators to use to find a type of element, is a learning process of using the Appium tool.

In this chapter, we will look at all the different locator strategies in detail, to find the mobile application elements in both iOS and Android, as well as to perform gesture actions on these elements.

Structure

In this chapter, we will discuss the following topics:

- Element Locator Strategy
 - Finding Element by UIAutomator
 - Finding Element by AccessibilityID
 - Finding Element by ID
 - Finding Element by Class Name
 - Finding Element by Name
 - Finding Element by Xpath
- Find Elements Method
- Miscellaneous Driver Methods
- Android Keycodes
- Element Properties/Attributes
- Automating Gestures

- o Tap Gesture
- o Long Press Gesture
- o Swipe and Scroll Gesture

Objectives

This chapter will help us understand the basics of finding mobile application elements using locators and how to perform actions and gestures on them. In modern applications, there are advanced mobile-specific functionalities such as click-to-call, barcode scanning, embedded GPS, and location service depending on the business, and so on. It comes down to mobile elements that make the building components of mobile applications, and locating these elements is the fundamental step in automating mobile applications. Elements can be a button, text fields, input fields, and so on. Once the elements are uniquely identified, we will perform actions such as tapping, swiping from bottom to top, long press, and so on. These are called gestures in mobile applications.

Element locator strategy

The most significant and potent component of Appium is Inspector, which makes items in mobile applications easier to identify and more efficient. Some of the

features include clicking elements and inspecting, a hierarchy view of elements with ID, Xpath, and so on, displayed separately for easier element locating, finding element coordinates that help locate elements using their position, and so on. There are different locator strategies available in Appium, and some are specific to platforms. Selenium web driver locator strategies such as a CSS selector, link text, partial link text, and tagname are not supported by the Appium API because the mobile page source (DOM) is not XML-based. However, Appium supports a subset of these Selenium WebDriver locator strategies, such as finding elements by class Name and Xpath.

The order of locators to be considered is ID/Accessibility ID, Name, Value, Class Name, Xpath. Let us look at some of these dominantly used locators in Appium.

Finding element by UIAutomator

UIAutomator is the Android native locator strategy to identify the element. It uses the UISelector methods to locate the elements, and all the methods of UISelector are detailed in the link: <https://developer.android.com/reference/kotlin/ccessi/test/>

uiAutomator/UiSelector

There are UISelector methods that can identify the elements using the index number, resourceId, text, class name, and so on. In *Figure 6.1*, the element is identified using **ANDROID_UIAUTOMATOR** locator and UISelector method “index” with index number passed as the argument. This set the search criteria to match the element by the node index in the layout.

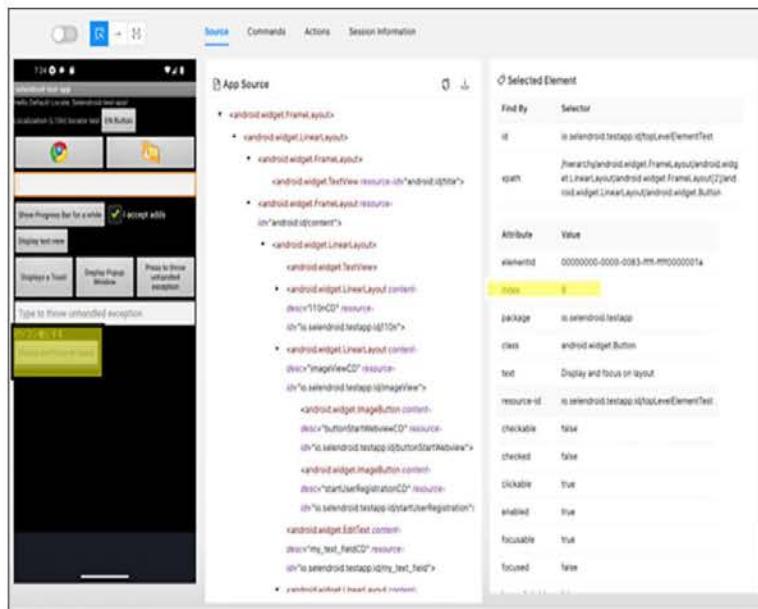


Figure 6.1: Element with Index

The following code includes recap on how to trigger the Android application.

CODE

```
1. from appium import webdriver  
2. from appium.webdriver.common.appiumby import AppiumBy  
  
3. desired_capabilities={'platformName':'Android','platformVersion':  
   '13','deviceName': 'Pixel 6 Pro API 33','app': '/Users/  
yogashivamathivanan/Downloads/selendroid-test-app.apk',  
   'appPackage': 'io.selendroid.testapp','appActivity': 'io.  
selendroid.testapp.HomeScreenActivity'}  
4. driver = webdriver.Remote('http://localhost:4723/wd/hub', desired_  
   capabilities)  
5. displayAndFocusLayout = driver.find_element(AppiumBy.ANDROID_
```

```

    UIAUTOMATOR, "UISelecto().index(9)")

6. displayAndFocusLayout.click()

7. driver.quit()

```

Finding Element by AccessibilityID

This is the most used and preferred locator and should be given higher priority than other locators if available. This locator is dominantly used for cross-platform scripts which work in both iOS and Android applications. *Figure 6.2* features the element with AccessibilityID:

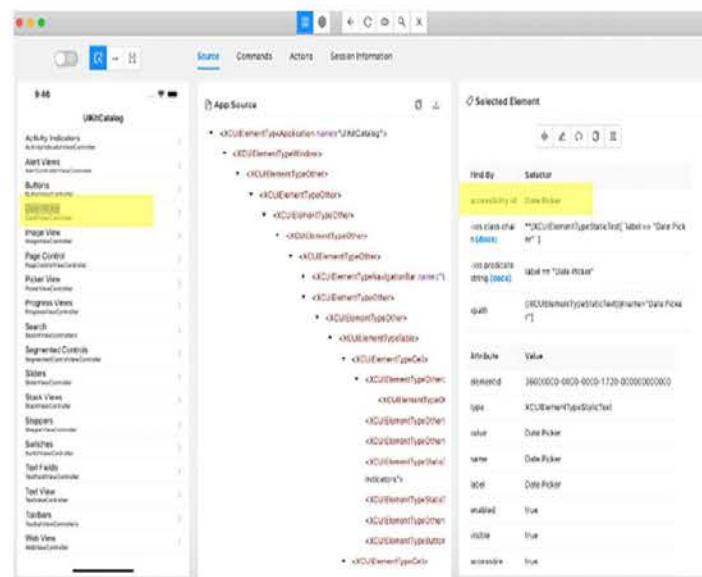
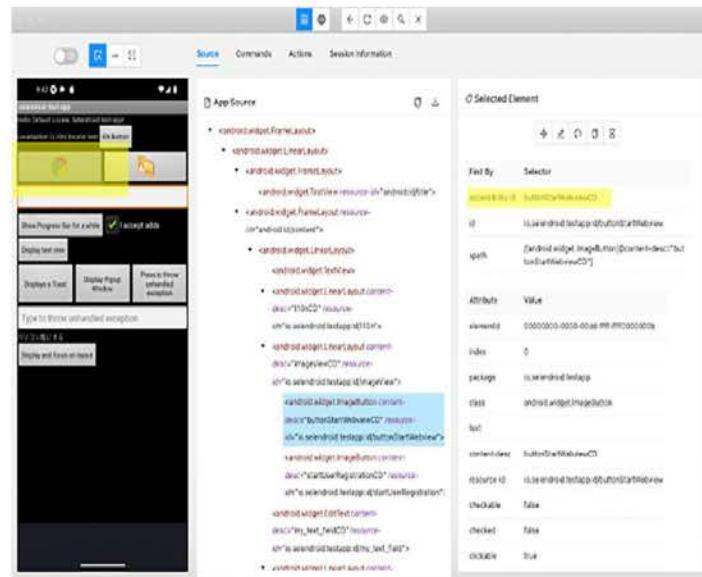


Figure 6.2 • Element with AccessibilityID

The following code includes recap of how to trigger iOS application.

CODE

iOS

1. from appium import webdriver
2. from appium.webdriver.common.appiumby import AppiumBy
- 3.
4. desired_caps = {'platformName': 'IOS', 'platformVersion': '15.5', 'deviceName': 'iPhone 13 Pro Max', 'automationName': 'XCUITest', 'app': ('/Users/yogashivamathivanan/Library/Developer/Xcode/DerivedData/UIKitCatalog-dskplzeeekmyvnwtcjosyjblrks/Build/Products/Debug-iphonesimulator/UIKitCatalog.app')}
5. driver = webdriver.Remote("http://127.0.0.1:4723/wd/hub", desired_caps)
6. datePicker=driver.find_element(AppiumBy.ACCESSIBILITY_ID,"DatePicker")
7. datePicker.click()

Android

1. button=driver.find_element(AppiumBy.ACCESSIBILITY_ID,"buttonStart-WebviewCD")
2. button.click()

Finding Element by ID

This is the simplest locator and each element has a unique ID to easily identify the element and perform the action. ID is considered cross-platform, which means that it works similarly in an IOS and Android, making the script usable in both operating systems. In Android, the ID is “**resource-id**” attribute. In iOS, there is no concept of ID as in Android, but we can use **accessibilityID** instead. We have seen examples of finding elements by ID, in *Chapter 4, Appium For Mobile Automation*.

Finding Element by Class Name

Class Name does not ensure finding elements uniquely, because multiple elements in a page can have the same class name but can be combined with other locators

such as index value, ID, text, and so on, which helps find a particular element. For iOS, Class Name is denoted by the XCUI element's complete name, which starts with **XCUIElementType**. Examples include **UIAButton** and **UIARadioButton**, whereas in Android, the full name of the UIAutomator2 class is stated in the class name like **android.widget**, as shown in the following *Figure 6.3*:

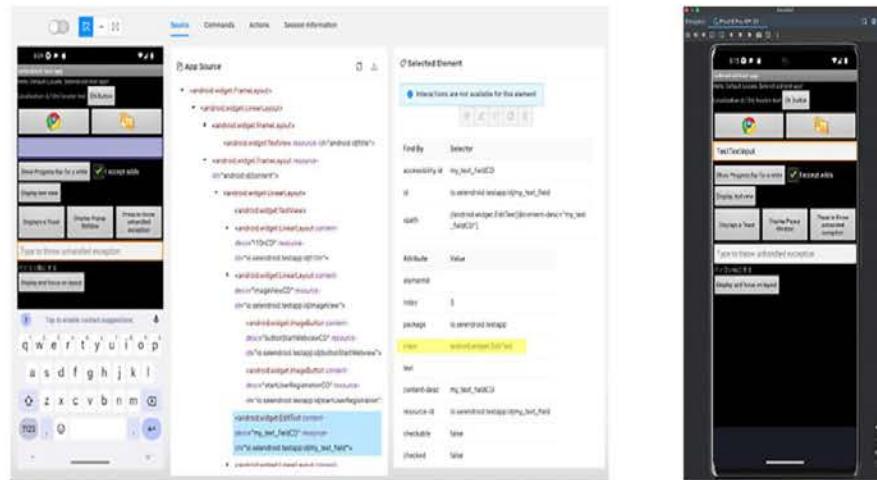


Figure 6.3: Finding Element with class Name

CODE

- buttonElement = driver.find_element(AppiumBy.CLASS_NAME, "android.widget.EditText")
- buttonElement.send_keys("TestTextInput")

Finding Element by name

This is a common locator to locate the element. Refer to *Figure 6.4*:

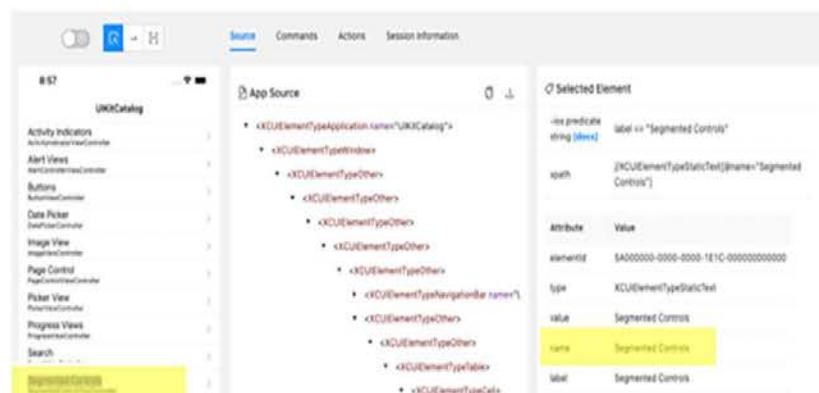




Figure 6.4: Finding Element with Name

CODE

```
1. segmentElement = driver.find_element(AppiumBy.NAME,"Segmented Controls")  
2. segmentElement.click()
```

Finding Element by Xpath

Like Xpath in Selenium, which examines the XML structure of the app to find the elements, Xpaths are helpful for locating dynamic elements. Xpath locators are not cross-platform and unreliable, because the path to find the element may change with the addition of new elements (try to use relative Xpath). It has performance delay and takes a long time to locate the element, depending on the number of elements on the page, and it can frequently result in stale element exceptions. As a result, using the name, id, and AccessibilityID to locate the element, should be given higher priority. In *Figure 6.5*, the Xpath of the element can be retrieved easily without any effort using Appium inspector.

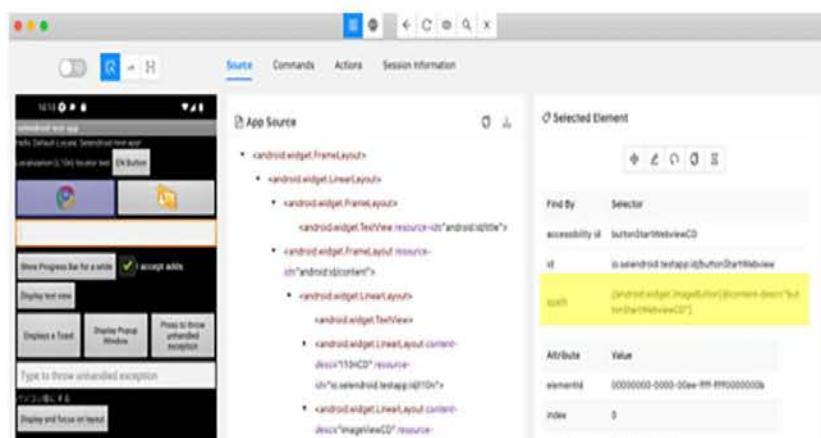




Figure 6.5: Finding Element with Xpath

CODE

```
1. buttonElement = driver.find_element(AppiumBy.XPATH, "//android.widget.ImageButton[@content-desc='buttonStartWebviewCD'])")
```

```
2. buttonElement.click()
```

Relative Xpath example -

```
1. driver.findElement(By.xpath("//XCUIElementTypeButton[@value='sample']"))
```

Find Elements Method

This is like **find element** method, but it returns list of elements with similar properties.

CODE

```
1. buttonList = driver.find_elements(AppiumBy.CLASS_NAME, "android.widget.Button")
2. print(len(buttonList))
```

Output - 7

Miscellaneous Driver Methods

There are certain driver methods that can be utilized to find the state and application information. For example, the application package and activity can be gathered using command “**adb shell dumpsys window windows**” as discussed in Appium Desired Capabilities section, in *Chapter 4, Appium For Mobile Application*. But with driver, we can find these properties. Similarly, application context, orientation, location, locked status, and so on, can also be retrieved.

CODE

```
1. print(driver.getPackageName())
```

```
1. print(driver.current_activity)
2. print(driver.current_package)
3. print(driver.current_context)
4. print(driver.orientation)
5. print(driver.location)
6. print(driver.is_locked())
```

Output

```
1. .HomeScreenActivity
2. io.selendroid.testapp
3. NATIVE_APP
4. PORTRAIT
5. {'latitude': 37.4217937, 'longitude': -122.083922, 'altitude': 5e-324}
6. False
```

Android Keycodes

Android has some constants that uniquely identify the ASCII values of the device keypress. In simple terms, key codes are numeric values that correspond to physical or touch keys in the mobile keyboard. For example, 67 --> "KEYCODE_DEL", 4 --> "KEYCODE_BACK", 66 --> "KEYCODE_ENTER", and so on. The list of complete keycode is given in the link: <https://developer.android.com/reference/android/view/KeyEvent.html>.

In the following code, after opening the application, click on the element, send text “Shiva” in the text box. Keycode 67 will remove character “a” leaving “Shiv” in the text box. Finally, click on back button using keycode 4.

CODE

```
1. from appium import webdriver
2. from appium.webdriver.common.appiumby import AppiumBy
3.
```

```

4. desired_capabilities={'platformName':'Android','platformVersion':
    '13', 'deviceName': 'Pixel 6 Pro API 33','app': '/Users/
yogashivamathivanan/Downloads/selendroid-test-app.
apk','appPackage': 'io.selendroid.testapp','appActivity': 'io.
selendroid.testapp.HomeScreenActivity'}

5. driver = webdriver.Remote('http://localhost:4723/wd/hub', desired_
capabilities)

6. driver.find_element(AppiumBy.ACCESSIBILITY_ID,
    "startUserRegistrationCD").click()

7. usernameElement = driver.find_element(AppiumBy.ID, "io.selendroid.
testapp:id/inputUsername")

8. usernameElement.send_keys("Shiva")

9. usernameElement.click()

10.driver.press_keycode(67)

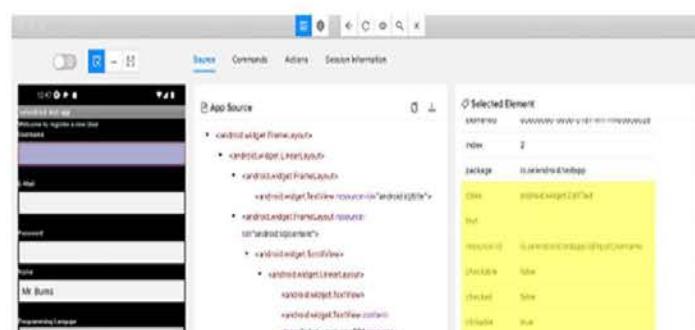
11.driver.press_keycode(4)

12.driver.quit()

```

Element Properties/Attributes

We can use element methods to retrieve the element attributes or properties. In the following code, will retrieve some of the attributes/properties, in the *Figure 6.6*:



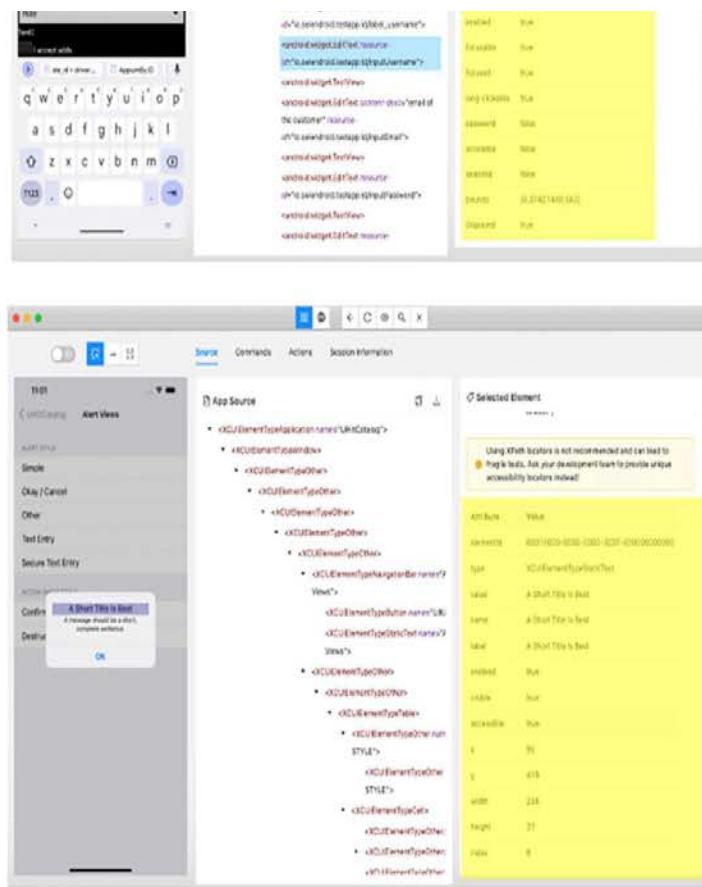


Figure 6.6: Finding Element attributes/properties

CODE

1. `driver.find_element(AppiumBy.ACCESSIBILITY_ID,"startUserRegistrationCD").click()`
2. `usernameElement = driver.find_element(AppiumBy.ID, "io.selendroid.testapp:id/inputUsername")`
3. `print(usernameElement.get_attribute("focusable"))`
4. `print(usernameElement.get_attribute("class"))`

5. `print(usernameElement.is_enabled())`
6. `print(usernameElement.is_selected())`
7. `print(usernameElement.is_displayed())`
8. `driver.quit()`

Output

1. true
2. android.widget.EditText
3. True
4. False
5. True

CODE - IOS

```
1. from appium import webdriver
2. from appium.webdriver.common.appiumby import AppiumBy
3.
4. desired_caps = {'platformName': 'IOS', 'platformVersion': '15.5',
   'deviceName': 'iPhone 13 Pro Max',
   'automationName': 'XCUITest', 'app': (
5.           '/Users/yogashivamathivanan/Library/Developer/Xcode/
   DerivedData/UIKitCatalog-dskplizeekmyvnbtcjosyjblrks/
   Build/Products/Debug-iphonesimulator/UIKitCatalog.app')}
6.
7.
8. driver = webdriver.Remote("http://127.0.0.1:4723/wd/hub", desired_
   caps)
9. driver.find_element(AppiumBy.ACCESSIBILITY_ID, "Alert Views").click()
10. driver.find_element(AppiumBy.ACCESSIBILITY_ID, "Simple").click()
11.
12. alertElement = driver.find_element(AppiumBy.XPATH, "//XCUIElementTypeStaticText[@name='A Short Title Is Best']")
13. print(alertElement.get_attribute("type"))
14. print(alertElement.get_attribute("index"))
15. print(alertElement.get_attribute("label"))
```

```
16. print(alertElement.is_enabled())
17. print(alertElement.location)
18. print(alertElement.text)
19.
20. output
```

```
20. output  
21. XCUIElementTypeStaticText  
22. false  
23. A Short Title Is Best  
24. True  
25. {'x': 95, 'y': 419}  
26. A Short Title Is Best
```

Automating Gestures

So far, we have seen the fundamentals of Appium automation, such as identifying and clicking on a button or entering text into a text field. However, the mobile applications allow advanced interactions with the UI using gesture. Gesture is an action performed in mobile using hand/finger. Gestures include pinch zoom in and zoom out, tapping, double tapping, long pressing, drag and drop, swiping right to left, pulling up or down, and even multiple touches. Appium supports these Android gestures using Touch Actions class. Let us look at each of these gestures supported by Appium.

Tap Gesture

Tapping an element on the mobile screen, is like clicking an element in the web application. Similarly, in mobile application, tapping on login button in login screen is an example of tap gesture. We can tap knowing the X and Y coordinates of the element position in the UI. To find the coordinates, turn on the **Developer** option by navigating to **Settings | About Phone**. Tap on Build Number 5 times.

Now the Developer option is enabled on the phone/Simulator. Navigate to **System**, and you will see **Developer** options. Click on it and turn on **Pointer Location**, as shown in *Figure 6.7*. After enabling pointer location, there is x and y coordinate on the top of screen.


```

9.           'appActivity': 'io.selendroid.testapp.
HomeScreenActivity'}
10.
11.driver = webdriver.Remote('http://localhost:4723/wd/hub', desired_
capabilities)
12.actions = TouchAction(driver)
13.element = driver.find_element(AppiumBy.ID, 'io.selendroid.
testapp:id/input_adds_check_box')

```

Tap Using X and Y coordinates

```
14.actions.tap(x=800, y=900, count=1).perform()
```

Tap Using locating the element

```
15.actions.tap(element=element, count=1).perform()
```

Long press gesture

We must click on an element without releasing for a certain duration, because actions cannot be performed on these elements by tapping. It requests to keep pressing for certain amount of duration, to perform the intended action. Like tapping, long press can also be performed using element locator or by providing X and Y coordinates of the element.

CODE

```
1. longPressElement = driver.find_element(AppiumBy.ACCESSIBILITY_ID,
"buttonTestCD")
```

Long Press Using X and Y coordinates

```
2. actions.long_press(x=800, y=900, duration=5).perform()
```

Long Press Using locating the element

```
3. actions.long_press(longPressElement, duration=5).perform()
```

Swipe and scroll gesture

Swipe gesture is when tap is used in combination with move action across the screen, either from left to right, right to left, top to bottom or bottom to top. This can be achieved by finding the location to tap, then the location to release the tap

and duration of the swipe. Appium provides swipe method which takes arguments starting from X and Y coordinate of the page and ending on the X and Y coordinate of the page, along with the time it will take to swipe.

With the swipe gesture, we can also achieve scrolling. If element is not in the page view to perform an action, you must scroll to position the element to view, before performing any action. To confirm that the element is in view, we can use swipe in combination with element property **isDisplayed()**. This can be accomplished simply by using scroll method as well. Let us look at the code to perform swipe and scroll (using swipe and scroll method) gesture.

CODE

```
1. from appium import webdriver
2. from appium.webdriver.common.appiumby import AppiumBy
3.
4. desired_capabilities = {'platformName': 'Android',
5.                         'platformVersion': '13', 'deviceName':
6.                         'Pixel 6 Pro API 33',
7.                         'app': '/Users/yogashivamathivanan/
8.                         Downloads/Android_Demo_App.apk',
9.                         'appPackage': 'com.code2lead.kwad',
10.                        'appActivity': 'com.code2lead.kwad.
11.                          MainActivity'}
12.
13. deviceSize = driver.get_window_size()
14. screenWidth = deviceSize['width']
15. screenHeight = deviceSize['height']
16. print('width: ' + str(screenWidth) + 'height: ' + str(screenHeight))
17. #Swipe right to left
```

```
18. driver.swipe(start_x=screenWidth*.9,      start_y=screenHeight*.5,
   end_x=screenWidth*.1, end_y=screenHeight*.5, duration=5000)
19. #Swipe left to right
20. driver.swipe(start_x=screenWidth*.1,      start_y=screenHeight*.5,
   end_x=screenWidth*.9, end_y=screenHeight*.5, duration=5000)
21. #Swipe bottom to top
22. driver.swipe(start_x=screenWidth*.5,      start_y=screenHeight*.8,
   end_x=screenWidth*.5, end_y=screenHeight*.3, duration=5000)
23. #Swipe top to bottom
24. driver.swipe(start_x=screenWidth*.5,      start_y=screenHeight*.3,
   end_x=screenWidth*.5, end_y=screenHeight*.8, duration=5000)

25. #Example of Scroll until the element is displayed with for loop
26. for _ in range(10):
27.     try:
28.         value= driver.find_element(AppiumBy.ID,"elementID").is_
   displayed()
29.         if value is True:
30.             break
31.     except:
32.         driver.swipe(340, 1500, 3400, 1000, 5000)
33.     continue

34. driver.find_element(AppiumBy.ID, "elementID").click()
```

CODE – SCROLL IN IOS

```
6.          '/Users/yogashivamathivanan/Library/Developer/Xcode/
DerivedData/UIKitCatalog-dskplzeekmyvnbtjosyjblrks/
Build/Products/Debug-iphonesimulator/UIKitCatalog.app')}

7.

8. driver = webdriver.Remote("http://127.0.0.1:4723/wd/hub", desired_
caps)

9. original_ele = driver.find_element(AppiumBy.ACCESSIBILITY_ID, "Text
View")

10.destination_ele = driver.find_element(AppiumBy.ACCESSIBILITY_ID,
"Alert Views")

11.driver.scroll(original_ele, destination_ele)

12.destination_ele.click()
```

Conclusion

Appium locators and gestures are the base of mobile automation. This helps in identifying elements uniquely and performing actions on them. We have seen different actions and gestures that can be performed on the identified element and how to leverage the power of Appium inspector to make our life easier in accomplishing the task. Having a strong hold on this topic is necessary to automate native apps, mobile web apps, and hybrid apps. The next step in Appium is to explore the automation of hybrid application, parallel execution, switching between apps and then looking at synchronization issues and framework design.

Key facts

- Appium extends Selenium so that the element locators in mobile can be used in Appium in similar fashion as used in Selenium.
- Appium inspector can be used to locate the element and its properties.
- Appium major locators include Finding Element by UIAutomator, AccessibilityID, ID, Class Name, Name and Xpath.
- Special action in mobile applications is called gestures, and that includes tap gesture, long press, swipe and scroll gesture.
- Pointer location in developer option gives the X and Y coordinate of any location in the application, which helps in locating the element when it

cannot be identified uniquely.

Questions

1. What are the advantages of Appium Inspector?
2. What are element locator strategies in Appium?
3. What are Gestures in Appium and how is it different from Actions performed in Web Application.
4. What is the difference between Swipe and Scroll methods?

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



CHAPTER 7

Synchronization, Exception Handling and Assertions

Introduction

Selenium and Appium are used to automate web and mobile applications, but common challenges must be addressed while working with either application. We need to ensure that the test flows from beginning to end without any interruptions, for robust and reliable test automation, and anticipated interruptions are handled to ensure the flow. The necessity to address this situation is unreliable and varying testing conditions exists in every test execution. For instance, the text element that needs to be validated is present in DOM but is not visible yet because of, say network latency, causing the "**ElementNotVisibleException**", or the element state changed and it no longer appears on the DOM in the middle of execution, causing "**StaleElementException**". The test is required to be instructed in the middle of the execution, to wait for a certain amount of time, by introducing delays or handling the expected exceptions before the test results in a false failure. Introducing delay ensures that the test execution speed matches the application speed. The goal is to

allow the Selenium or Appium test to fail because of the functional error caused, but the test should not fail because of any such exceptions.

Each significant step of the execution needs validation in place, to check whether the received output is equal to the expected output. For example, when we go to

the application URL and perform a login, the test needs to get and confirm the title of the page, that validates if the user credentials are valid and login was successful. If there is failure, the test should stop the execution and report the login incorrect failure. In another case, the requirement is to collect the total number of links on a given page, but the test should not stop execution after the incorrect number. We need to have validation in place, in crucial execution points and control over the flow of execution. Even after validation fails, the test should be able to accomplish, stop or continue the execution, depending on the severity of the failure. This can be achieved by using assertions which is a function or method that returns true or false based on the element validation condition. The assertion is used to validate a single unit of complete test execution, also called a unit test, which allows the validation check at the single unit test level.

Structure

In this chapter, we will discuss the following topics:

- Synchronization
 - Unconditional Synchronization
 - Conditional Synchronization
 - Implicit Wait
 - Explicit Wait
 - Fluent Wait
- Selenium Exceptions and Handling
 - Exceptions
 - Exception Handling
- Assertion
 - Pythons Assert Statement

Objectives

In this chapter, we will see in detail the different synchronization methods in Selenium and common exceptions in Selenium, as well as the handling of these exceptions. Selenium WebDriver offers implicit wait, and explicit wait to address the synchronization exceptions. We will go over how to implement these waits within our test. The exceptions can be handled using a try-catch block. The assertions are used to add validations to the condition of the elements, that allow confirmation of the expected element behavior with the actual observed behavior.

Synchronization

In general terms, synchronization is when two or more components work together to perform any action, the components need to run in sync to achieve the expected output. Interaction and actions performed on the application under test are possible with multiple background processes which can take varied response times to complete depending on various conditions for example increased server response time. The browser or native mobile can take some time to populate the element on UI depending on the response time, user actions such as page refresh, and so on, and Selenium tries to find or perform an action on the element before it appears on the screen resulting to exceptions, thus a false test failure.

The test scripts should be written to make the test automation tool and application under test operate at the same desired pace. Synchronization can be classified into two categories – **Unconditional Synchronization (Static Wait)** and **Conditional Synchronization (Dynamic wait)**.

Unconditional Synchronization

This indicates a static wait time that suspends the test execution until the specified wait time is elapsed. In Python, we use `time.sleep([time in seconds])` to achieve static wait. The static wait can be used when we know the approximate wait time for the response or process to complete, especially when working with external systems where it is not possible to write conditions to check the system response status. This is not suggested as the execution will wait irrespective of the early or late response affecting the overall execution time.

Conditional synchronization

This indicates the dynamic wait time that suspends the test execution until the condition is satisfied, and throws an exception only if the expected condition is not met after the specified time. There are mainly two kinds of waits in conditional synchronization: Implicit Wait and Explicit Wait.

Implicit wait

An implicit wait instructs the web driver to locate the element in DOM for a specified amount of time, before throwing an exception, and proceeds to the next step if it finds the element before the specified time. This is a global wait that applies to all the instances of web driver locating the elements on the page during the execution. Hence, implicit wait is set for the entire duration of the test. In Python, implicit wait

is set using the `implicitly_wait()` method of the driver, as shown in the following code. In the following code, the WebDriver also waits for 30 seconds before throwing `NoSuchElementException`.

CODE

```
1. import time
2. from selenium import webdriver
3. from selenium.webdriver.chrome.service import Service
4. from selenium.webdriver.common.by import By
5. from webdriver_manager.chrome import ChromeDriverManager
6.
7. driver = webdriver.Chrome(service=Service(ChromeDriverManager("").install()))
8.
9. driver.implicitly_wait(time_to_wait=30)
10.
11. driver.get("https://magento.softwaretestingboard.com/customer/account/login")
12.
13. time.sleep(15) # This Halts the execution for 15 seconds -
    Unconditional Synchronization
14.
15. driver.find_element(By.XPATH, "//*[@id='email' and @type='email']").
    send_keys("admin12@gmail.com")
16. driver.find_element(By.XPATH, "//input[@id='pass' or @
    title='password']").send_keys("admin12!")
17. driver.find_element(By.XPATH, "//fieldset[@class='fieldset login']//"
    span[contains(text(),'Sign In')]).click()
18. driver.find_element(By.XPATH, "//span[text()='Men']").click()
19. driver.find_element(By.XPATH, "//input[@type='email']").send_
    keys("adminadmin@gmail.com")
20. driver.find_element(By.XPATH, "//*[contains(text(),'Subs')]").click()
21.
```

```

22. #Random text to find - The Webdriver waited for 30 seconds implicit
   wait before throwing NoSuchElementException

23.

24. driver.find_element(By.XPATH, "//a[text()='TopsRandom!@#$']").
   click()

25. driver.quit()

```

OUTPUT

1. raise exception_class(message, screen, stacktrace)
2. selenium.common.exceptions.NoSuchElementException: Message: no such
 element: Unable to locate element: {"method":"xpath","selector":"//
 a[text()='TopsRandom!@#\$']"}
3. (Session info: chrome=107.0.5304.121)

Explicit wait

An explicit wait is a code written specifically for an element and for a certain condition to occur before proceeding to the next step. These conditions may vary from element to element. The explicit wait defines WebDriver wait, which defines the wait time and expected conditions, which are pre-defined WebDriver methods. For example, `visibility_of_element_located`, `alert_is_present` and other supported conditions, are shown in *Figure 7.1*.

In the following code, the condition `invisibility_of_element` is expected. The `emailElement` does not go invisible, causing a timeout exception. In Explicit wait, the exceptions can be handled within the web driver wait by using the argument “`ignored_exceptions`” and we can define the frequency to check the condition using the argument `poll_frequency`, as in line 13. Adding a poll frequency to check for the condition at a regular interval is called Fluent Wait.

Fluent wait

The Fluent wait allows the automation script to wait until an element condition is satisfied on the web page by repeatedly polling the webpage at a specified interval. The benefit of using Fluent wait is that it provides a more flexible and intelligent way of waiting for the condition to occur. The fluent wait can be customized to suit specific needs, by adjusting the polling interval, the maximum wait time, and the conditions to be met. This helps to make the automation script more efficient, reliable, and faster.

CODE

```
1. from selenium import webdriver
2. from selenium.webdriver.chrome.service import Service
3. from selenium.webdriver.common.by import By
4. from webdriver_manager.chrome import ChromeDriverManager
5. from selenium.webdriver.support.ui import WebDriverWait
6. from selenium.webdriver.support import expected_conditions as ec
7. from selenium.common import TimeoutException
8.
9. driver = webdriver.Chrome(service=Service(ChromeDriverManager("").install()))
10. driver.get("https://magento.softwaretestingboard.com/customer/account/login")
11. emailElement = driver.find_element(By.XPATH, "//*[id='email' and @type='email']")
12. elementWait = WebDriverWait(driver, 30).until(ec.invisibility_of_element(emailElement))
13. #elementWait = WebDriverWait(driver, 5, ignored_exceptions=[Exception], poll_frequency=2).until(ec.invisibility_of_element(emailElement))
14. emailElement.send_keys("admin12@gmail.com")
15. driver.quit()
```

OUTPUT

```
1. Traceback (most recent call last):
2.   File "/Users/SeleniumDemo9.py", line 15, in <module>
3.     elementWait = WebDriverWait(driver, 30).until(ec.invisibility_of_element(emailElement))
4.   File "/Users/Selenium/venv/lib/python3.10/site-packages/selenium/webdriver/support/wait.py", line 90, in until
5.     raise TimeoutException(message, screen, stacktrace)
6. selenium.common.exceptions.TimeoutException: Message:
```

Similarly, explicit wait can be implemented in mobile application program, as in the following code.

CODE

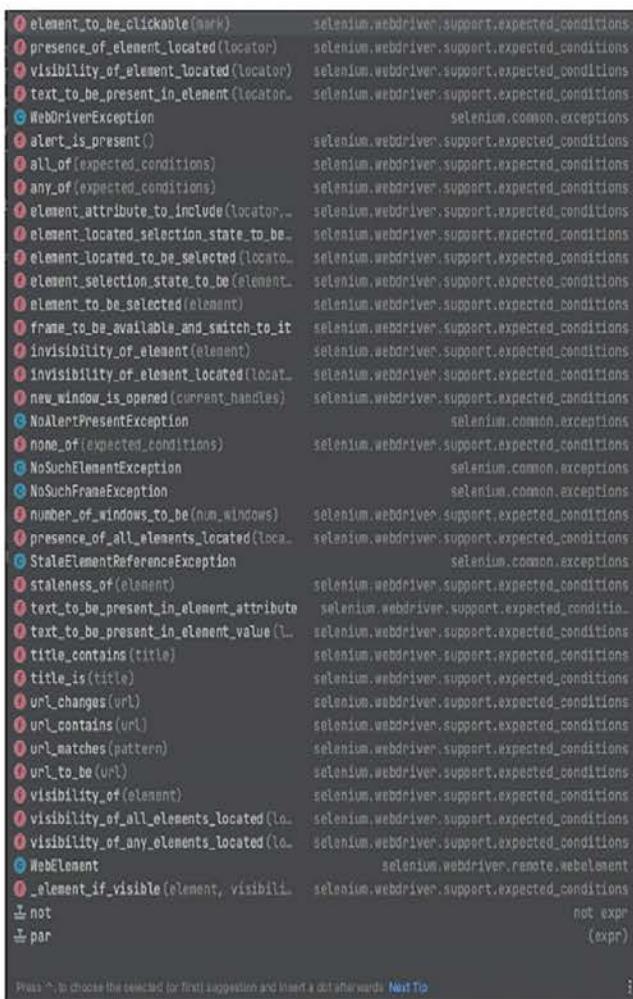
```
1. from appium import webdriver
2. from appium.webdriver.common.appiumby import AppiumBy
3. from selenium.common import ElementNotVisibleException,
   NoSuchElementException
4. from selenium.webdriver.support.wait import WebDriverWait
5. from selenium.webdriver.support import expected_conditions as ec
6.
7. desired_capabilities = {'platformName': 'Android',
8.                         'platformVersion': '13', 'deviceName':
   'Pixel 6 Pro API 33',
9.                         'app': '/Users/yogashivamathivanan/
   Downloads/Android_Demo_App.apk',
10.                        'appPackage': 'com.code2lead.kwad',
11.                        'appActivity': 'com.code2lead.kwad.
   MainActivity'
12. driver = webdriver.Remote('http://localhost:4723/wd/hub', desired_
   capabilities)
13.
14. explicitWait = WebDriverWait(driver, 25, poll_frequency=1,
15.                               ignored_exceptions=[ElementNotVis-
   ibleException, NoSuchElementException])
16. element = explicitWait.until(ec.presence_of_element_located((Ap-
   iumBy.ID, "com.testapp.sel:id/Value")))
17. element.click()
18. driver.quit()
```

OUTPUT

```
19. selenium.common.exceptions.TimeoutException: Message:
20. Stacktrace:
21. NoSuchElementException: An element could not be located on the page
```

using the given search parameters.

Refer to Figure 7.1:



The screenshot shows a code editor with a list of explicit wait expected conditions. The list is organized into two columns: the method name and its corresponding module. The methods are color-coded with red numbers and blue symbols. The modules are also color-coded. The list includes methods like element_to_be_clickable, presence_of_element_located, visibility_of_element_located, text_to_be_present_in_element, and many others. The modules listed are selenium.webdriver.support.expected_conditions, selenium.common.exceptions, and selenium.webdriver.remote.webelement.

Method	Module
element_to_be_clickable(locator)	selenium.webdriver.support.expected_conditions
presence_of_element_located(locator)	selenium.webdriver.support.expected_conditions
visibility_of_element_located(locator)	selenium.webdriver.support.expected_conditions
text_to_be_present_in_element(locator,	selenium.webdriver.support.expected_conditions
WebErrorException	selenium.common.exceptions
alert_is_present()	selenium.webdriver.support.expected_conditions
all_of(expected_conditions)	selenium.webdriver.support.expected_conditions
any_of(expected_conditions)	selenium.webdriver.support.expected_conditions
element_attribute_to_include(locator,	selenium.webdriver.support.expected_conditions
element_located_selection_state_to_be_	selenium.webdriver.support.expected_conditions
element_located_to_be_selected(locat..	selenium.webdriver.support.expected_conditions
element_selection_state_to_be(element,	selenium.webdriver.support.expected_conditions
element_to_be_selected(element)	selenium.webdriver.support.expected_conditions
frame_to_be_available_and_switch_to_it	selenium.webdriver.support.expected_conditions
invisibility_of_element(element)	selenium.webdriver.support.expected_conditions
invisibility_of_element_located(locat..	selenium.webdriver.support.expected_conditions
new_window_is_opened(current_handles)	selenium.webdriver.support.expected_conditions
NoAlertPresentException	selenium.common.exceptions
none_of(expected_conditions)	selenium.webdriver.support.expected_conditions
NoSuchElementException	selenium.common.exceptions
NoSuchFrameException	selenium.common.exceptions
number_of_windows_to_be(num_windows)	selenium.webdriver.support.expected_conditions
presence_of_all_elements_located(loc..	selenium.webdriver.support.expected_conditions
StaleElementReferenceException	selenium.common.exceptions
staleness_of(element)	selenium.webdriver.support.expected_conditions
text_to_be_present_in_element_attribute	selenium.webdriver.support.expected_condit..
text_to_be_present_in_element_value (l..	selenium.webdriver.support.expected_conditions
title_contains(title)	selenium.webdriver.support.expected_conditions
title_is(title)	selenium.webdriver.support.expected_conditions
url_changes(url)	selenium.webdriver.support.expected_conditions
url_contains(url)	selenium.webdriver.support.expected_conditions
url_matches(pattern)	selenium.webdriver.support.expected_conditions
url_to_be(url)	selenium.webdriver.support.expected_conditions
visibility_of(element)	selenium.webdriver.support.expected_conditions
visibility_of_all_elements_located(lo..	selenium.webdriver.support.expected_conditions
visibility_of_any_elements_located(lo..	selenium.webdriver.support.expected_conditions
WebElement	selenium.webdriver.remote.webelement
_element_if_visible(element, visibility)	selenium.webdriver.support.expected_conditions
not	not expr
or	(expr)

Figure 7.1: Explicit wait expected conditions

Selenium exceptions and handling

Let us now learn more about Selenium Exceptions and Handling.

Exceptions

Exceptions are unexpected and unintended disruptive behavior that interrupts the execution of the test script. Each exception is caused by a specific reason and terminates the test case execution. Hence, it requires to be handled to continue with

the uninterrupted flow of the execution. Exceptions thrown by the application can be handled. On the other hand, errors are created by environments and cannot be

handled. Selenium provides several built-in exceptions and we will learn how to handle them. The most common exceptions are “**NoSuchElementException**” which occurs when the web driver cannot locate the element, “**InvalidElementStateException**” when the element is disabled, and so on. Let us look in more detail at other common Selenium exceptions:

- **NoSuchWindowException**: When the action needs to be performed in a browser window, the control is first switched to the window and then the action is performed. This exception is raised if the window is not present/loaded, and actions are performed on it. Line 13 in the following code will throw this exception because such a window handle does not exist.
- **NoSuchFrameException**: This exception occurs when the WebDriver is switching to an invalid or unavailable iframe, or the information does not match the available frames, and so on. Line 14 in the following code will throw this exception because of an invalid frame identifier.
- **NoSuchAlertException**: Similar to frame and window exceptions, this occurs when the expected alert is not present. Line 12 in the following code will throw this exception because the expected alert is not popped up.

CODE

```
1. from selenium import webdriver
2. from selenium.webdriver.chrome.service import Service
3. from selenium.webdriver.common.by import By
4. from webdriver_manager.chrome import ChromeDriverManager
5.
6. driver.find_element(By.XPATH,    "//*[@id='email']").send_keys("admin12@gmail.com")
7. driver.find_element(By.XPATH, "//*[@id='pass' or @title='password']").send_keys("admin12!@")
8. driver.find_element(By.XPATH, "//fieldset[@class='fieldset login']//span[contains(text(),'Sign In')]").click()
9. #driver.switch_to.alert.accept() - NoSuchAlertException
10.#driver.switch_to.window("1234567890") - NoSuchWindowException
11.#driver.switch_to.frame("0") - NoSuchFrameException
```

Here are a few exceptions:

- **ElementClickInterceptedException**: This exception occurs when the element is concealed or hidden. Click the element if it cannot be executed correctly.
- **ElementNotSelectableException**: The exception occurs when web element is present in the DOM but cannot be selected. This is applied mostly to radio button, checkbox, and so on, which requires to be selected.
- **ElementNotVisibleException**: The element is present in the web page but not visible because of hidden property in the html tag or requires prerequisite action to be performed.
- **InvalidElementStateException**: When the element state is disabled, for example, if you click on the disabled submit button, it is only enabled when all the required fields are entered.
- **NoSuchCookieException**: The exception is raised when there is no match for the cookie defined, within the cookies present in the active document of the browser's current context.
- **StaleElementReferenceException**: This exception is raised when an element that was previously found on the page is no longer attached to the DOM. This can happen due to the page being refreshed, navigated to a different URL, switched to a different window handle, or the element being updated dynamically.
- **TimeoutException**: The exception is raised when an operation times out, typically due to an element taking too long to appear or become intractable.

Exception Handling

Exception Handling is an important aspect of writing reliable and robust automation scripts. When interacting with web pages, we can encounter various exceptions as seen in the last section. It is common to occur due to reasons such as network connectivity, server errors, error working with web elements, and more. Thus, it is important to handle these exceptions to ensure the automation scripts continue to execute as expected, and in the event of failure, it provides us information to debug the error.

The Exceptions can be handled using a try-except block.

Syntax:

try:

Code that may raise an exception.

except:

Code to handle the raised exception.

The exception “**NoSuchElementException**” in the following code is handled using try and catch block. When handled, even with the exception thrown, the execution flow did not stop and the program was completed. Thus, exception handling should only be applied to logic that is independent and does not have any impact on the execution steps. For example, if the login operation is handled, there is no point in proceeding to the next step, in case of invalid login credentials.

Moreover, to handle unsure exceptions or chances of multiple exceptions, we can use **Exception**, which is the parent of all exceptions, as in line 19 in the following code.

CODE

```
1. from selenium import webdriver
2. from selenium.common import TimeoutException, NoSuchElementException
3. from selenium.webdriver.chrome.service import Service
4. from selenium.webdriver.common.by import By
5. from webdriver_manager.chrome import ChromeDriverManager
6.
7. driver = webdriver.Chrome(service=Service(ChromeDriverManager("").install()))
8. driver.get("https://magento.softwaretestingboard.com/customer/account/login")
9.
10. emailFieldName = driver.find_element(By.XPATH, "//div[contains(-
    text(), 'email address')]/following-sibling::div[1]/label/span").text
11.
12. try:
```

```
13.     emailTextElement = driver.find_element(By.XPATH, "//*[@  
        id='email#@#']")  
14. except NoSuchElementException:  
15.     print("No Such Element Exception happened")
```

```
16.  
17. try:  
18.     raise NoSuchElementException  
19. except Exception:  
20.     print("Exception is parent of all exception")  
21.  
22. driver.find_element(By.XPATH,    "//*[@id='email']").send_keys("ad-  
min12@gmail.com")  
23. driver.find_element(By.XPATH, "//*[@id='pass' or @title='pass-  
word']").send_keys("admin12!@")  
24. driver.find_element(By.XPATH, "//fieldset[@class='fieldset login']//  
span[contains(text(),'Sign In')]").click()  
25. driver.quit()
```

OUTPUT

1. No Such Element Exception happened
2. Exception is parent of all exception

Assertions

Assertions are the validation points in the script which validate the actual result with the expected results. It means that the verification is done to check if the state of the application is the same as what we are expecting as per the requirement. In the absence of an assertion, there is no option for determining if the test case has passed or failed. **If-else** statements can be used to compare expected and actual results, but this simply fixes the logging problem; the test will not fail unless you check the if-else condition logs. Thus, the assertion is used to create execution reports, that inform the developers about any errors or bugs associated with the test case. The assertions have no impact on the test case if it completes all the test steps, although they are reported if the test case fails. Assertions also help in debugging the failure when the test case fails.

The Assertion function or method gives two results: True or False (Pass or Fail). It passes only when the expected condition matches the actual result. When it fails, the **AssertionError** exception is raised. An assertion is used to test a single unit of application; hence it is called a unit test.

Python Assert Statement

Python comes with an inbuild assert statement that can be used in the programs for assertion conditions. A condition in an assert statement is one that should always be true. Assert terminates the program and throws an **AssertionError** if the condition is false and gives assertion error with the provided assertion error message.

In the following syntax for using Assert in Python, the error message is optional but recommended to easily debug in case of failure. Let us look at a few code examples.

assert <condition>, <error message>

- The assert in the following code is to validate that the page title contains “**LoginInvalid**”. Since the login page does not have a keyword exception, it is thrown as in the output.

CODE

```

1. from selenium import webdriver
2. from selenium.webdriver.chrome.service import Service
3. from webdriver_manager.chrome import ChromeDriverManager

4. driver = webdriver.Chrome(service=Service(ChromeDriverManager("").install()))
5. driver.get("https://magento.softwaretestingboard.com/customer/account/login")
6. assert "LoginInvalid" in driver.title,"The Title does not contain Login"
7. driver.quit()

```

OUTPUT

```

8.     assert "LoginInvalid" in driver.title, "The Title does not
contain Login"
9. AssertionError: The Title does not contain Login

```

- The following assertion validates the links on the page, and the assertion can be added by comparing the number of links with the expected number or adding an assert condition within the **if** condition. Moreover, in the list with all the link elements, we can add an assert to check if an element is part of the list or not, as in the following code, with the output as if the assertion fails.

CODE

```
1. from selenium import webdriver
2. from selenium.webdriver.chrome.service import Service
3. from webdriver_manager.chrome import ChromeDriverManager

4. driver = webdriver.Chrome(service=Service(ChromeDriverManager("").install()))
5. driver.get("https://magento.softwaretestingboard.com/customer/account/login")
6. assert "LoginInvalid" in driver.title,"The Title does not contain Login"
7. driver.quit()
```

OUTPUT

```
8. assert 49 == noOfLinks, "No of Links are no as expected"
9. AssertionError: No of Links are no as expected

10. assert False
11. AssertionError

12. assert emailTextElement in LinksElement, "The forgot your password link is not a part of the page."
13. AssertionError: The forgot your password link is not a part of the page.

14. assert forgotYourPasswordLink not in LinksElement, "The email Text element is not a link"
15. AssertionError: The email Text element is not a link
```

- In the preceding example, the execution will stop if the assertion fails. We can handle the exception using a **try-catch** block, as discussed in exception handling. Moreover, list and dictionary can also be asserted, as shown in the following code.

CODE

```
1. try:  
2.     num = 34  
3.     num2 = 0  
4.     assert num2 != 0, "Number 2 Cannot be equal to 0"  
5.     print(num/num2)  
6. except AssertionError:  
7.     print("Please Provide non zero number")  
  
8. list1 = [1, 2, 3, 4, 4]  
9. list2 = [1, 2, 3, 4, 4]  
10.list3 = [1, 2, 3, 4, 4]  
11.list4 = [1, 2, 3, 4, 4]  
12.list5 = [1, 2, 3, 4, 5]  
13.  
14.assert list1 == list2 == list3 == list5, "All lists are not equal"  
15.  
16.dict1 = {"List One": list1,  
17.           "list two": list2}  
18.dict2 = {"List One": list3,  
19.           "list two": list5}  
20.  
21.assert dict1 == dict2, "the two dict are not equal"
```

OUTPUT

```
22.Please Provide non zero number  
23.assert list1 == list2 == list3 == list5, "All lists are not equal"  
24.AssertionError: All lists are not equal  
25.  
26.assert dict1 == dict2, "the two dict are not equal"  
27.AssertionError: the two dict are not equal
```

Conclusion

The synchronization issue is resolved using waits, and this makes sure that the environmental issues or other factors preventing the action on the element are not affecting the execution. We looked at the implicit wait and explicit wait supported by Selenium with Python. Implicit wait tells the webdriver to wait for a certain time before throwing an exception, and explicit wait is specific to each element and waits for a certain condition to be fulfilled before throwing an exception. This chapter also introduced exceptions, their significance, and how to handle them. We discussed the most common exceptions in Selenium and the reason for their occurrence with examples. Moreover, the concept of assertions is introduced in this chapter. The assertions provide the validation points in the program to confirm that the application's expected behavior is matching with the actual behavior. In the next chapters, we will go into detail about how to bring all the concepts we learned so far together, and we will put these concepts in a framework.

Key facts

- One of the crucial lines of code that runs in a test case in Selenium is the wait statements. Selenium Waits makes the pages less vigorous and reliable.
- Selenium throws an '**ElementNotVisibleException**' message when you tend to locate an element present in your script, which is still not loaded on the web page.
- The main purpose of implicit Wait is to instruct the web driver to hold off on throwing an exception. Its default value is set to zero. The driver will wait for the duration set.
- Explicit Waits are also known as Dynamic Waits and wait for the specified condition to be fulfilled for the duration set before throwing an exception.
- Exceptions are unexpected and unintended disruptive behavior that interrupt the execution of the test script.
- Exceptions can be handled using a try-catch block.
- In coding, assertions are Boolean conditions or Boolean expressions that are always expected to be true. Assert statements have a condition and an optional message.

Questions

1. What is synchronization in web and mobile automation, and how is it addressed?
2. What are the different types of waits in Selenium with Python?
3. What are different types of Exceptions and how to handle them?
4. What are an Assertion and its significance?

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



CHAPTER 8

Hybrid Application Automation & Launching Multiple Apps

Introduction

Hybrid Applications are apps that are a combination of both native and web apps built to run on multiple platforms, unlike native applications, which are built specifically for a defined platform. Hybrid applications have common code bases developed using web languages such as HTML, CSS, and so on, to deliver iOS and Android applications. This is the major benefit of Appium, as it has inbuild hybrid support, using a chrome driver for Android applications that allows automation of any Chrome-backed Android web views. This does not require changing/switching the app to test it. Hybrid applications can be tested similarly to Selenium for web apps.

Along with the Hybrid application automation, we will see some advanced Appium functionalities such as automating two apps at the same time by switching between apps during the execution, which allows performing an action in a different native

app, and switching back to the application under test. For example, in a scenario where the main application sends an SMS code, we need to switch to messaging application to retrieve the code and paste it into the main application. This can be achieved by switching between the apps.

Structure

In this chapter, we will discuss the following topics:

- Hybrid Application
 - Hybrid Mobile App Examples
 - Webview
 - Android Hybrid App Automation
 - IOS Hybrid App Automation
- Switching between multiple applications

Objectives

So far, we worked with native apps and how to use Appium methods and tools to automate the native app flow. In this chapter, we will discuss the automation of hybrid apps in both android and iOS devices, and see how we can leverage the Selenium WebDriver utilities to achieve this. Moreover, we will also look at how we can switch between apps during the execution.

Hybrid application

Let us define different mobile applications in detail. Native apps are built in a specific programming language for a specific platform, either Android or iOS. Native iOS apps are written in Swift or Objective C and native Android apps are written in Java. Native apps can be downloaded from the platform's app store and cannot be accessed using web browser. A web app is an application built using HTML, CSS, and JavaScript that can be accessed using a web browser either from the desktop or from mobile phones. Hybrid applications can also be downloaded from the app store such as a native app, but it is a web app on the inside that are built using HTML, CSS, and JavaScript.

As in *Figure 8.1*, the hybrid application uses a middle layer that makes the hybrid application platform independent:

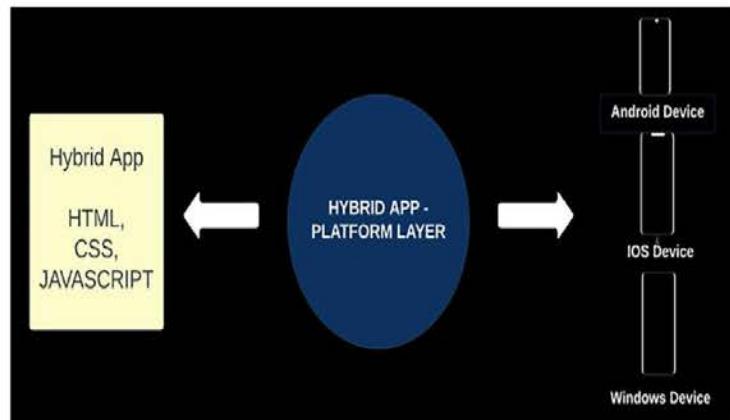


Figure 8.1: Cross-Platform Hybrid Applications

Hybrid mobile app examples

Some of the best-known and widely used hybrid app application examples are as follows:

- **Instagram:** A widely used photo and short video sharing social networking app is a Hybrid application. The hybrid approach helps users use the media even when the user is offline.
- **Gmail:** The most common email-sending application across the world uses a hybrid application model to design the app. By using the hybrid app, the performance and loading time have improved.
- **Uber:** The ridesharing and food delivery app is a hybrid application. The hybrid model helps the app with easier navigation and a smooth user interface.
- **Twitter:** A microblogging and social networking app is also a hybrid app, that helps the app to tackle high load and traffic.

WebView

A WebView can be defined as an embedded browser within a native app to display web content. In simple terms, WebView is a browser engine that loads web content inserted into the native app like an Iframe. WebView App is composed mostly of web technologies such as HTML, CSS, and JavaScript that make up web pages/user interfaces. WebView gives the web experience within the native app and makes it a hybrid application.

We can say that a WebView is simply a visual component that we would use to create the visuals of the native app. When using a native app, a WebView may be present among other native UI elements without realization.

In the following *Figure 8.2*, the WebView app downloaded from the play store has an embedded Google web application:

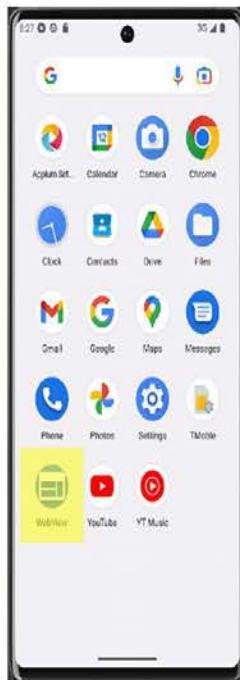


Figure 8.2: WebView app downloaded from the app store

Figure 8.3 shows the WebView of the Google web page content within the app:

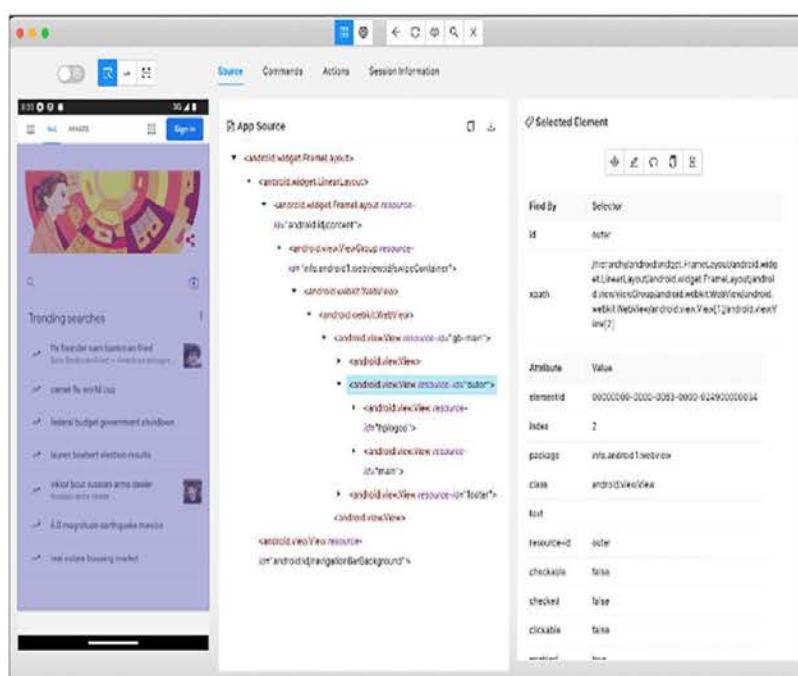


Figure 8.3: WebView embedded in native app

The following code opens the app shown in *Figure 8.2*. The first step in hybrid app automation is to identify the WebView context of the application. To interact with the WebView, we need to switch the driver control to the WebView context. Line 18 in the following code prints the available contexts of the app, as in the output, and it is a list with native app context '**NATIVE_APP**' and WebView context "**WEBVIEW_info.android1.webview**".

To interact with WebView, the driver needs to switch to WebView context as in line 19, and then we can interact with the element within WebView.

CODE

```
1. from appium import webdriver
2. from appium.webdriver.common.appiumby import AppiumBy
3. from selenium.common import ElementNotVisibleException,
   NoSuchElementException
4. from selenium.webdriver.support.wait import WebDriverWait
5. from selenium.webdriver.support import expected_conditions as ec

6. desired_capabilities = {'platformName': 'Android',
   'platformVersion': '13', 'deviceName':
   'Pixel 6 Pro API 33',
7.   'app': '/Users/yogashivamathivanan/Downloads/
   WebView_3.3.5_Apkpure.apk',
8.   'appPackage': 'info.android1.webview',
9.   'appActivity': 'info.android1.webview.MainActivity'}
10.
11.driver = webdriver.Remote('http://localhost:4723/wd/hub', desired_
   capabilities)

12.explicitWait = WebDriverWait(driver, 25, poll_frequency=1,
13.                           ignored_exceptions=[ElementNotVis-
   ibleException, NoSuchElementException])
14.acceptButton1 = explicitWait.until(ec.presence_of_element_
   located((AppiumBy.ID, "android:id/button1")))
15.acceptButton1.click()
```



```
16. acceptButton2      =      explicitWait.until(ec.presence_of_element_
located((AppiumBy.ID, "android:id/button1")))

17. acceptButton2.click()

18. print(driver.contexts)

19. driver.switch_to.context("WEBVIEW_info.android1.webview")

20. driver.quit()
```

OUTPUT

```
1. ['NATIVE_APP', 'WEBVIEW_info.android1.webview']
```

Android Hybrid App Automation

With a complete understanding of WebView, once the driver is switched to the WebView context, the driver needs to interact with the elements in WebView. As mentioned, the WebView contents are mainly HTML, CSS, and JavaScript. We can interact with the elements like we do with web pages using Selenium web driver. In the web application automation, we had DOM and a browser element inspector to locate and identify the elements. We can use the chrome browser to identify the elements.

Identifying WebView elements

Let us take an example of a chrome application, which is a hybrid application and open apple.com website on the Emulator device, as shown in *Figure 8.5*. Open the Chrome browser on the machine and type “`chrome://inspect/#devices`” in the browser URL. This should show the Emulator device connected and opened chrome application info, as shown in the following *Figure 8.6*. Clicking on **Inspect** opens the dev tools as shown in *Figure 8.7*. This helps identify the elements uniquely.

Chrome application package name and app activity can be retrieved using command `adb shell dumpsys window windows`, as shown in *Figure 8.4*:


```

Xcode[108]: 108:12:07:07:11, UnderOS[11]: UnderOS[11]: com.
r0l0gicalConfiguration:(i:0 318rcs26merc [en_10] idtr swkido wklido h8z8do 500dp1 rml long pert finger evertv/v v dso/v v mounds:Rect(0, 0 - 1440, 3120) nApp
ounds:Rect(0, 136 - 1440, 3036) nMaxBounds:Rect(0, 0 - 1440, 3120) nDisplayRotation:ROTATION_0 vwindowMode fullscreen vDisplayWindowMode fullscreen nActivityType:under
lined nAlwaysOnTop:undefined nIteration:nROTATION_0 i:38 fontWeight:4fjustment:-4)
mFocusable:0 mFocused:0 mFocusedRect:0x0
mImeTarget in display 0 Window@a44254 v0 com.google.android.apps.chrome.Main
mInputTarget in display 0 Window@a44254 v0 com.google.android.apps.chrome.Main
mControlTarget in display 0 Window@a44254 v0 com.google.android.apps.chrome.Main
mMinTaskSizeOfDisplay# 220 mInTouchMode: true
mIsVisible: true
mLastDisplayFreezeDuration# 250ms due to Window@c75d9c v0 com.google.android.apps.nexuslauncher.NexusLauncherActivity
mIsWakeLockAcquired: false mWakeLockOwner: null
mHighResTaskSnapshotScale# 0.8
mTaskSnapshotTasked: true
mSnapshotCache
mInputMethodWindowHandle# 0x0 InputMethod
mReversalShouldFalse
mHoldScreenWindowNull
mOscuringWindowNull
mRotatedBordered: true mDisplayEnabled: true
mTransactionSequence# 1238
mDisplayFrozen: false window# 0 client: false app# 0 mOrientation# 0 mLastOrientation# -1
waitingForConfigFile
Animation settings: disabled false window# 0 transition# 1.0 animator# 1.0
yogesh@yogesh-MacBook-Pro ~ % 

```

Figure 8.4: Chrome Native application package and activity

Appium Inspector Desired capabilities for Android Chrome is as follows:

1. {
2. "appium:appPackage": "com.android.chrome",
3. "appium:appActivity": "com.google.android.apps.chrome.Main",
4. "platformName": "Android",
5. "appium:deviceName": "Pixel 6 Pro API 33",
6. "appium:udid": "emulator-5554"
7. }

Figure 8.5 features the Chrome Native application with web page content:

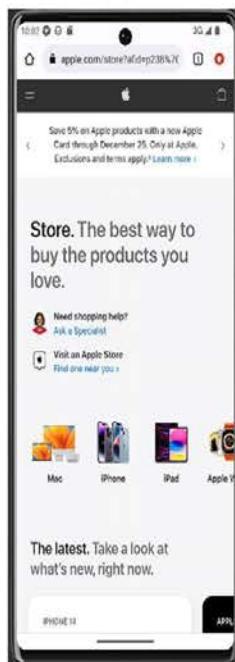


Figure 8.5: Chrome Native application with web page content

Figure 8.6 features the Chrome Browser emulator info with WebView content to inspect:

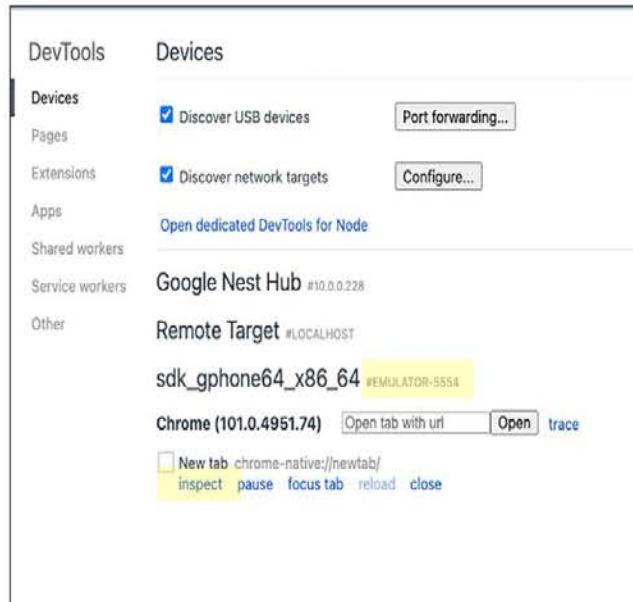


Figure 8.6: Chrome Browser Emulator info with Webview content to inspect

Figure 8.7 features the Chrome Dev Tools for WebView Contents:

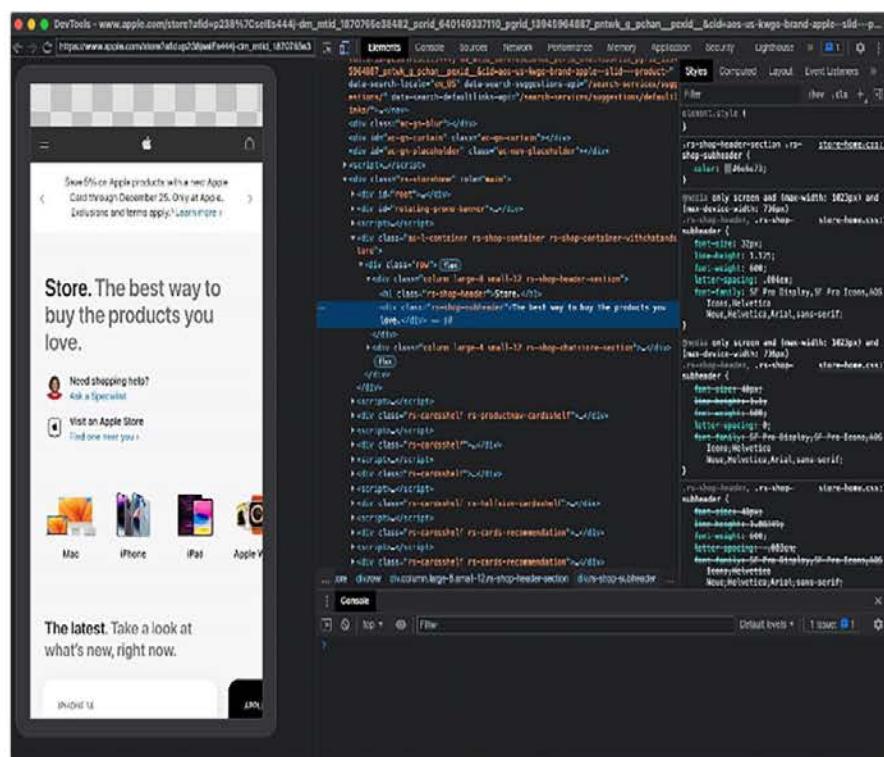


Figure 8.7: Chrome Dev Tools for WebView Contents

Switching to WebView Context to perform the required action and switching back to Native App

Once the WebView context is identified, we can obtain the available contexts using `driver.contexts`, which returns a list. We can loop through the list and switch to WebView context as in the following code.

In the following code, we are opening the Chrome application, visiting the Apple.com webpage and switching the driver context to the apple web application using loop in line 33, and performing a test to verify the latest iPhone using assert as in line 42. After the test, we are switching back to the Chrome native app and now visiting Samsung.com webpage as in line 46 and 50 respectively.

CODE

```
1. from appium import webdriver
2. from appium.webdriver.common.appiumby import AppiumBy
3. from selenium.common import ElementNotVisibleException,
   NoSuchElementException
4. from selenium.webdriver.support.wait import WebDriverWait
5. from selenium.webdriver.support import expected_conditions as ec
6.
7. desired_capabilities = {'platformName': 'Android',
8.                         'platformVersion': '13', 'deviceName':
   'Pixel 6 Pro API 33',
9.                         'app': '/Users/yogashivamathivanan/Downloads/
   selendroid-test-app.apk',
10.                        'appPackage': 'com.android.chrome',
11.                        'appActivity': 'com.google.android.apps.
   chrome.Main'}
12.
13. driver = webdriver.Remote('http://localhost:4723/wd/hub', desired_
   capabilities)
14.
15. explicitWait = WebDriverWait(driver, 25, poll_frequency=1,
16.                           ignored_exceptions=[ElementNotVis-
   ibleException, NoSuchElementException])
```

```
17. acceptButton      =      explicitWait.until(ec.presence_of_element_
   located((AppiumBy.ID, "com.android.chrome:id/terms_accept")))
18. acceptButton.click()
19.
20. noThanksButton    =      explicitWait.until(ec.presence_of_element_
   located((AppiumBy.ID, "com.android.chrome:id/negative_button")))
21. noThanksButton.click()
22.
23.
24. searchBox         =      explicitWait.until(ec.presence_of_element_
   located((AppiumBy.ID, "com.android.chrome:id/search_box_text")))
25. searchBox.click()
26.
27. urlButton         =      explicitWait.until(ec.presence_of_element_
   located((AppiumBy.ID, "com.android.chrome:id/url_bar")))
28. urlButton.click()
29. urlButton.send_keys("https://www.apple.com")
30. driver.press_keycode(66)
31. appContexts = driver.contexts
32. print(driver.contexts)
33. for appType in appContexts:
34.     if appType == "WEBVIEW_chrome":
35.         driver.switch_to.context(appType)
36.         break
37.
38. learnMoreLink      =      explicitWait.until(ec.presence_of_element_
   located((AppiumBy.XPATH, "//a[@data-analytics-title='Learn more
   about iPhone 14']")))
39. learnMoreLink.click()
40. latestIphoneEle   =      explicitWait.until(ec.presence_of_element_lo-
   cated((AppiumBy.XPATH, "//a[@data-analytics-title='product in-
```

```
aex ] )))  
41.latestIphone = latestIphoneEle.text  
42.assert latestIphone == "iPhone 14", "The latest iphone is incorrect"
```

```
43.buylatestIphoneEle = explicitWait.until(ec.presence_of_element_  
located((AppiumBy.XPATH, "//a[@class='ac-ln-button']")))  
44.buylatestIphoneEle.click()  
45.  
46.driver.switch_to.context("NATIVE_APP")  
47.  
48.urlButton = explicitWait.until(ec.presence_of_element_  
located((AppiumBy.ID, "com.android.chrome:id/url_bar")))  
49.urlButton.click()  
50.urlButton.send_keys("https://www.samsung.com")  
51.driver.quit()
```

OUTPUT

```
1. ['NATIVE_APP', 'WEBVIEW_chrome']
```

Troubleshooting of possible error related to chrome version

In the case of the error **No Chrome driver found that can automate Chrome version due to incompatible chrome version in the emulator and appium server** appearing, the issue can be resolved by downloading the compatible Chrome version matching the emulator and passing the path within desired capabilities.

Navigate to <https://chromedriver.chromium.org/downloads> and download the compatible Chrome version. Verify the compatibility by checking the emulator Chrome version in the settings. Add the downloaded path to desired capability, as follows:

```
1. 'chromedriverExecutable':'/Users/[userName]/Downloads/  
chromedriver'
```

IOS Hybrid App Automation

iOS Hybrid application automation does not require any setup as in Android. In the iOS application to interact with the WebView, Appium establishes a connection with the simulator using a custom debugger. So once additional desired capabilities are added, we can directly interact with WebView without having to switch to WebView.

As in the following code, the desired capabilities `autoWebView` and `browserName` is set to “true” and “safari” respectively, and allow to interact with the WebView directly without having to switch the context. In the following code, when the

contexts are printed, we see the output has WebView context, but can interact with the element without switching driver context. Additionally, WEBVIEW in iOS has a random number attached to it every execution.

CODE

```
1. from appium import webdriver
2. from appium.webdriver.common.appiumby import AppiumBy
3. from selenium.common import ElementNotVisibleException,
   NoSuchElementException
4. from selenium.webdriver.support import expected_conditions as ec
5. from selenium.webdriver.support.wait import WebDriverWait
6.
7. desired_caps = {'platformName': 'IOS',
8.                  'platformVersion': '15.5',
9.                  'deviceName': 'iPhone 13 Pro Max',
10.                 'autoWebview': 'true',
11.                 'browserName': 'safari',
12.                 'automationName': 'XCUITest'}
13.
14. driver = webdriver.Remote("http://127.0.0.1:4723/wd/hub", desired_
   caps)
15. explicitWait = WebDriverWait(driver, 25, poll_frequency=1,
16.                               ignored_exceptions=[ElementNotVis-
   ibleException, NoSuchElementException])
17. driver.get("https://www.apple.com")
18. print(driver.contexts)
19. learnMoreLink = explicitWait.until(ec.presence_of_element_
   located((AppiumBy.XPATH, "//a[@data-analytics-title='Learn more
   about iPhone 14']")))
20. learnMoreLink.click()
21. driver.quit()
```

OUTPUT

```
1. ['NATIVE_APP', 'WEBVIEW_2311.1']
```


Appium Inspector Desired capabilities for iOS Safari are as follows:

```
1. {
2.   "platformName": "IOS",
3.   "appium:platformVersion": "15.5",
4.   "appium:deviceName": "iPhone 13 Pro Max",
5.   "appium:automationName": "XCUITest",
6.   "browserName": "safari"
7. }
```

As shown in *Figure 8.8*, the element attributes can be obtained using the Appium inspector application:

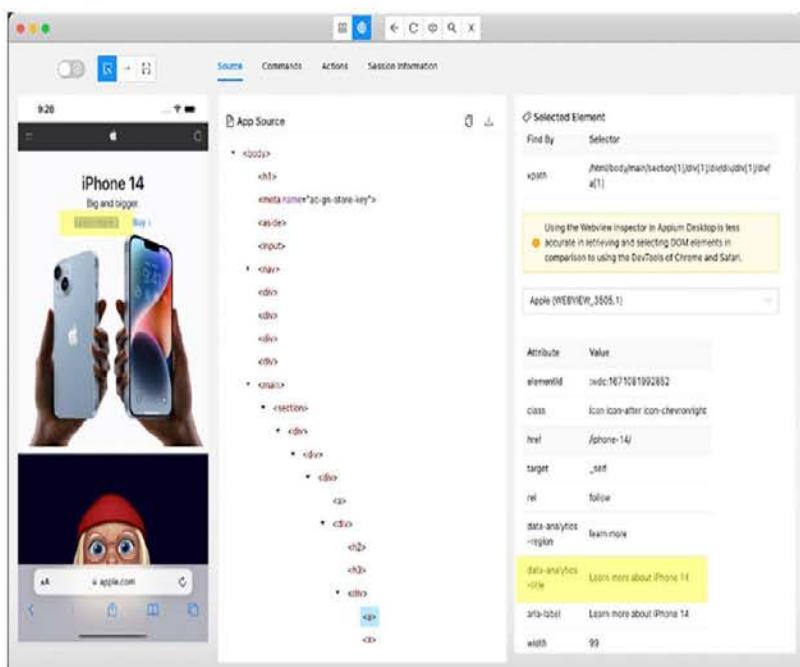


Figure 8.8: Appium inspector inspecting web view contents

Switching between multiple applications

Let us now see how to automate a scenario that involves multiple applications in succession in a single session. Let us automate the flow where we open the application under test and perform the initial action. We will then switch to the settings application and check where the internet is connected and switch back to our application for the next set of actions.

Open the Setting application in the emulator and find the package name and activity name by typing `adb shell dumpsys window windows` in the terminal. In the following code, the settings application package name is `com.android.setting` and the activity name is `com.android.settings.Settings`. The application can be switched by using the driver method `start_activity` and passing the argument package name and activity name of the application. Please refer to the following figure:

CODE

```
1. from appium import webdriver
2. from appium.webdriver.common.appiumby import AppiumBy
3. from selenium.common import ElementNotVisibleException,
   NoSuchElementException
4. from selenium.webdriver.support.wait import WebDriverWait
5. from selenium.webdriver.support import expected_conditions as ec
6.
7. desired_capabilities = {'platformName': 'Android',
8.                         'platformVersion': '13', 'deviceName':
   'Pixel 6 Pro API 33',
9.                         'app': '/Users/yogashivamathivanan/
   Downloads/WebView_3.3.5_Apkpure.apk',
10.                        'appPackage': 'info.android1.webview',
11.                        'appActivity': 'info.android1.webview.
   MainActivity'}
12.
13. driver = webdriver.Remote('http://localhost:4723/wd/hub', desired_
   capabilities)
14.
15. explicitWait = WebDriverWait(driver, 25, poll_frequency=1,
16.                               ignored_exceptions=[ElementNotVis-
   ibleException, NoSuchElementException])
17.
18. acceptButton1      =      explicitWait.until(ec.presence_of_element_
   located((AppiumBy.ID, "android:id/button1")))
19. acceptButton1.click()
```

```
20.  
21. acceptButton2      =      explicitWait.until(ec.presence_of_element_  
located((AppiumBy.ID, "android:id/button1")))  
22. acceptButton2.click()  
23.  
24. driver.start_activity("com.android.settings", "com.android.  
settings.Settings")  
25.  
26. networkInternet = explicitWait.until(ec.presence_of_element_  
located((AppiumBy.ANDROID_UIAUTOMATOR, 'text("Network &  
internet")')))  
27. networkInternet.click()  
28. internet = explicitWait.until(ec.presence_of_element_  
located((AppiumBy.ANDROID_UIAUTOMATOR, 'text("Internet")')))  
29. internet.click()  
30. androidWifi = explicitWait.until(ec.presence_of_element_  
located((AppiumBy.ANDROID_UIAUTOMATOR, 'text("AndroidWifi")')))  
31. androidWifi.click()  
32. internetStatus = explicitWait.until(ec.presence_of_element_  
located((AppiumBy.ID, "com.android.settings:id/entity_header_  
summary")))  
33. internetConnStatus = internetStatus.text  
34. assert internetConnStatus== "Connected", "Internet is not connected"  
35.  
36. driver.start_activity("info.android1.webview", "info.android1.  
webview.MainActivity")  
37. print(driver.contexts)  
38.  
39. driver.switch_to.context("WEBVIEW_info.android1.webview")  
40.  
41. searchElement = explicitWait.until(ec.presence_of_element_  
located((AppiumBy.XPATH, "//*[@class='gLfyf']")))  
42. searchElement.click()  
43. searchElement.send_keys("https://www.apple.com")
```

44.

45. driver.quit()

OUTPUT

1. ['NATIVE_APP', 'WEBVIEW_info.android1.webview']

Conclusion

Hybrid application automation is one of the key features of Appium. Hybrid apps are unique in consisting of both native and web components. Hybrid apps are coded in the same technology as web and mobile web applications such as HTML, CSS, and JavaScript where web components are embedded in a native container on the mobile device. The two main contexts in a hybrid application are native app and WebView.

In this chapter, we have covered hybrid application automation using Appium, how to identify contexts, and switch to the desired context to perform the action. In addition, we have seen how to switch between applications in a succession in a single session. Now that we have gathered knowledge of Selenium and Appium tools and its components, the next step is to bring all the components together and create a framework. In the coming chapters, we will find out the most efficient way to automate using a framework, for both Selenium and Appium.

Key facts

- Appium allows the testing of hybrid applications without having to change the application.
- The Hybrid application can be automated in a similar way Selenium is used to automate web applications.
- The Hybrid application has native and web view contexts such as **NATIVE_APP** or **WEBVIEW_1**, and the driver control is required to switch to the desired context to perform the action on the elements in the context.
- A WebView is a web application embedded inside a native application.
- iOS hybrid application automation is less complicated compared to Android.
- All the contexts can be obtained using the **driver.contexts** method, and it can then be switched to the desired context using the **driver.switch_to_context("")** method.

- Switching between multiple apps in a single session can be done using the driver `start_activity("packageName", "activityName")`

Questions

1. What is a WebView in a Hybrid application?
2. How to switch to the desired context?
3. What desired capability is required to be added for the iOS hybrid application?
4. How to switch to a native app from a hybrid app using Appium?

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



CHAPTER 9

Selenium

Automation

Framework – Part 1

Introduction

Test Automation professionals must have the ability to build robust frameworks, where each framework should have the ability to handle all the application's scenarios. The basic flow of UI automation is to launch the application, input the data, perform and validate application flows, log the results, close the application, and report the results. Once this flow is automated, we have reduced the manual efforts and execution time significantly. An organization could have tens and thousands of test cases to be automated. In an Agile environment, it is crucial to deliver automation in a time efficient manner. Moreover, increasing complexities in web applications require the test script to evolve to perform uninterrupted flow execution, while integrating with multiple external and internal applications such as payment systems, order and shipments, databases, and so on, to parameterize test data, work with the operating system, along with file systems; to automate the application in different platforms and more. Considering these, it is not optimum to

write the script for each of the test cases individually, and thus there arises a need for a system that is organized and can leverage the already automated test steps to automate further test cases in a time efficient manner.

Now consider automating a flow that requires identifying web elements to perform a login operation and then validating scenario 1. Now, second test case also requires performing a login operation before performing scenario 2. Let us say that separate test scripts are written to automate both scenarios; if the login web element attribute used to identify the web element in the login page is updated or changed, then both scenario test scripts are required to be updated in order to accommodate the change and fix the failing test. If there are multiple scenarios implemented similarly, then all the test scripts that involve login will fail, and thus, it will be required to update the attribute in thousands of test cases involving login step, to fix the automation script. Since the scenarios have login operation as a common step, it is ideal to implement the login automation script once, and utilize the script for all the validations that require the login step. In this case, if the login web element attribute is updated, it will only require updating the script at one location. Hence, the framework is used to achieve code reusability. When the team is working on the automation effort, the framework also helps the team to stay focused on the format, steps, and rules tailored to the team's needs. For example, the login step in the preceding example can be implemented by one team member, and the framework helps the usage of the implemented login script across the team for different automation scenarios and maintain the standard format across the team.

High efficiency in the automation script can be achieved by utilizing the automation framework. The Automation framework is not a single tool; it is a collection of tools in sync, in order to support efficient automation of the application. In this chapter, we will deep dive into the test automation framework world and understand how to create scalable frameworks, different frameworks and implementations, and Python packages, that can be used to achieve the maximum benefits of automation framework.

Structure

In this chapter, we will discuss the following topic:

- Automation Framework

- Pytest
- Robot Framework
- Unittest
- Behave
- Difference Between Pytest, Robot Framework, Behave and Unittest

Objectives

We are well versed in the testing strategies to automate web applications, how to work with different web elements, identify them uniquely and create automation scripts to execute the flow. The framework design is the core of automation testing; here, we will put the knowledge from previous chapters and learn how to organize the test scripts to achieve time efficiency, maintainability, formatting, and re-usability using test automation frameworks. The Automation framework design is of different types and implementation can be different, depending on the type of application and test automation requirements. Hence, we will investigate each of the framework types and different Python packages that can be used to make the framework more robust and stable.

Automation framework

The automation framework is an organized set of elements such as standards, rules, protocols, best practices, and guidelines, in order to handle multiple automation files and tools systematically, to achieve efficient automation tasks. In the Automation framework, these elements are used to increase code reusability, reduce maintenance, and increase time efficiency. Organizations are not only testing the application, but the expectation is to do it as quickly and comprehensively as possible. Automation frameworks are a major part of the Agile software development and test model, especially for continuous integration in achieving progressive automation, that is, immediately automating the features as they are developed in the same sprint along with regressive automation, that is, automation of existing features to make sure the newly developed features are not breaking the existing functionalities. There are a lot of tools available in the market for test automation; all the tools can be leveraged using a single framework and see the results.

Why Automation framework?

As discussed in the introduction of this chapter, more efficiency in automation

scripts increases the chances of success in achieving test automation promptly in an agile environment. As the automated test suites grow over time, it makes automation a more complex task. Automation frameworks make the automation scripts organized, and well-designed, thus helping to manage the complexity of test automation by providing a structured approach to developing, executing, and maintaining automated tests.

Let us consider an example of a web e-commerce application that allows customers to purchase products online. There could be numerous features ranging from searching for the product, selecting and configuring the product, adding products to the cart, and adding shipping details along with billing information, checkout, and tracking of the product. Moreover, the application will require a variety of tests to ensure quality such as functional, regression, smoke, security, and so on. The automation framework can help here manage the complexity of the testing process and efficiently execute all the automated tests. The framework might include the following:

- **Tool/Library:** The framework might use a tool such as Selenium or a library like Cypress, to automate the UI of the application.
- **Test data management:** The framework might have a test data management system with a database, property file, or external CSV file to store and retrieve data.
- **Logging and Reporting:** The framework might use a mechanism to log and report the results of test execution.
- **Test case management:** The framework might include a way to maintain the test cases with priorities and grouping for easier execution of a specific set of test cases.
- **Integration with other tools:** The framework might be integrated with other tools or external applications.

Additionally, the automation framework can be improved parallelly to accommodate infrastructure change, newer integrations, or added application functionalities. In short, the automation framework serves as a comprehensive strategy that can guide the test automation effort of the project, towards the desired test automation results, and thus should be an integral part of the software testing automation process.

Benefits of Automation framework

Automation testing without a framework is not recommended even for a single module small application with limited test cases. The following are some of the key benefits that explain the necessity and growing importance of the test automation framework in the project:

- **Scalability and modularity:** Automation can be easily scaled to accommodate additional test cases for new features, test data sets, and environments. In an agile environment, as newer features are added to the application, the automation framework can also be modified and evolve along with the application to accommodate new test scenarios, dependencies, test data, and

pre-production & post-production environments. With the use of a modular testing framework, comprehensive test coverage can be achieved.

- **Maintenance and cost effectiveness:** Automation testing can save costs compared to manual testing, with time and resources involved in manual testing efforts. However, with an automation framework, once the optimum code coverage is achieved, there is no manual intervention required. Progressive test cases can be automated with slight modifications to the data sets, because of existing code and rules in the framework. Furthermore, reusing code in a test automation framework can reduce the time required to develop test cases and make maintenance easier.
- **Reusability:** The framework comes with an automation code database that can be reused repeatedly to speed up test development. The team can ensure consistency and reliability when reusing the automated tests. For example, test automation code for logging in, navigating to certain application pages, and connecting to the database to retrieve data, can be reused in multiple test cases involving this step.
- **Documentation:** With the unified process of building and executing test cases, an organized and well-developed automation framework can serve as documentation of the functionalities of the application.
- **Ease of scripting, and understandability:** The automation framework establishes process definitions that make sure that both individuals and teams have consistent coding practices. The framework consolidates the code from the automation team together to make sure no code is duplicated. Moreover, the framework helps and allows new team members to quickly understand the automation structure and contribute.
- **Integration:** Selenium can integrate with other tools and frameworks to provide end-to-end testing, including continuous integration and delivery. This allows teams to automate the entire software development and testing process ensuring that all the stages of the development process are tested.

process, ensuring that all the stages of the development process are tested and validated.

- **Reporting:** One of the major advantages of the automation framework is the ability to configure reporting tools as per the project requirement for detailed test results of the test execution, that helps the team to identify and prioritize defects/bugs.

Types of automation framework

Automation framework can be divided into Built-in Framework and customized/User defined Framework. Built-in frameworks such as pytest, robotframework, unittest, and so on, are already available in the market. Here, all the benefits we discussed for the automation framework are pre-configured; we just need to follow the syntax and utilize packages pre-defined in the framework to leverage the benefits. We will go over the implementation of these frameworks, but let us look at the various user-defined frameworks.

Linear scripting framework

As the name suggests, all the test steps are written and executed in sequential order, and it is also known as the record and playback framework. This approach involves recording user actions in the test script and playing them back. This kind of framework is ideal for small functions and applications, when time-saving is necessary, actions are linear and predictable, and tests are relatively simple. This type is not very flexible, and any changes in the application functionality require script modification. Hence, it is not ideal for large, complex projects.

Advantages:

The advantages of linear scripting framework are as follows:

- Linear scripting framework does not require complex custom coding, and hence, expertise in test automation is not necessary.
- The test scripts can be generated as the recording process is faster than coding.
- The framework is easy to follow due to its sequential recording process.

Challenges:

The challenges of linear scripting framework are as follows:

- The test data is hard coded and thus cannot rerun the test cases with multiple sets of data. Therefore, the script needs to be re-generated for different data sets.
- It cannot accommodate any changes in the application, which will require a lot of rework, and thus maintenance is difficult.

Modular testing framework

The modular framework breaks down the application into smaller manageable modules, which is performing specific features of the application. The automation script is written for each of these smaller modules such as logging in or adding products to the cart, and so on, making the framework flexible, reusable, and scalable. Therefore, it is suitable for large-scale testing projects.

Advantages

The advantages of modular testing framework are as follows:

- An update to the application will only impact one module implementation which should be modified.
- Because of modularization, the framework is cost-efficient and easy to maintain.

Challenges:

The challenges of modular testing framework are as follows:

- Data parameterization is not flexible, and data is hard coded. Thus, different module implementation is required for different data sets.
- Coding is required.

Data-Driven testing framework

The data is the focus of this framework. Data-driven framework separates test data and test script that enables the reuse of test scripts with different test data, which could be stored within the framework or external data source such as a database, files, or spreadsheets. This framework is ideal for applications working with a large

set of data.

Advantages

The advantages of data-driven testing framework are as follows:

- Data is not hard coded, and hence, multiple data sets can be used in the same test case. The same test case will be executed multiple times depending on the number of data records available in the data source.
- Because of the Data-driven nature of this framework, less code is required to perform multiple scenarios.

Challenges

The challenges of data-driven testing framework are as follows:

- Coding knowledge and high-test automation experience are required to develop such a framework to work with multiple data sources.
- Higher time to develop the framework.

Keyword-driven testing framework

The Keyword-Driven framework also known as table-driven testing, separates the test logic from the test scripts, using a set of keywords to define the test steps. This implies that the automation test script performed is based on the keywords specified in a specific project table and the keyword defines the actions on the elements, such as click, verify, compare, type text, and so on. This framework makes it easy to modify and maintain the test and is ideal for applications with test cases involving many conditional statements and decision-making.

Advantages

The advantages of keyword-driven testing framework are as follows:

- A single keyword can be used in multiple scripts making the code reusable.

Challenges

The challenges of keyword-driven testing framework are as follows:

- High setup cost, time-consuming and complex setup of the framework.
- Constant maintenance of keywords is required

Behavior-driven development framework

The Behavior driven framework or BDD framework, emphasizes collaboration between developers, testers, and product owners. BDD is easily understandable because it uses simple English.

Advantages

The advantages of behavior-driven development framework are as follows:

- Improves collaboration between technical and non-technical team members, thus shrinking the gap between them.
- Reusability of code with multiple test data.

Challenges

The challenges of behavior-driven development framework are as follows:

- The requirements are required to be well documented, which will impact the test scripts directly.

Hybrid testing framework

The hybrid testing framework combines one or more of the previously-mentioned frameworks and leverages multiple frameworks' advantages to meet project demands. This framework is ideal for complex projects that require multiple automation techniques.

Advantages

The advantages of hybrid testing framework are as follows:

- Depending on the project structure and demand, multiple frameworks can come together with their benefits.

Challenges

The challenges of hybrid testing framework are as follows:

- A skilled automation tester is required to set up the complex framework.
- High maintenance and cost to maintain multiple framework structures.

Choosing the right automation framework

Automation framework choice depends heavily on the project requirements and

needs. Following are the factors to consider:

- **Testing needs:** Before choosing the right framework, analyze the test automation candidates within the team. This includes the type of application under test, testing goals, test execution/flow and complexity, test environment, application integrations, test automation stakeholders, and test budget. Analyze if testing will be done in multiple applications, multiple platforms, and so on.
- **Analyze frameworks:** Based on the testing needs, different frameworks are required to be analyzed, that suit the team's automation requirements and applications technology stack, such as a programming language, OS, database, and so on.
- **Scalability and flexibility:** As new features are added to the application, the framework should be scalable to handle all the scenarios and flexible enough to accommodate the changes with minimum effort.

- **Evaluate the cost:** The framework's cost should align with the project budget and provide value.
- **Community support:** The framework should have enough user base to be able to get community support in case of questions or issues with the framework.

Pytest

We have a complete understanding of the framework and its importance in the Software Automation process. Let us deep dive into some dominant frameworks used in the industry. Pytest is one of the most popular and widely used open-source Python testing frameworks, that is simple and flexible to handle complex testing scenarios. Pytest is designed to be easy to use, yet powerful enough to handle complex testing scenarios. Pytest also has an active user community to support.

Some of the features of Pytest are:

- Pytest supports test automation of functional testing, unit testing, and integration testing.
- Pytest supports a rich set of plugins that extends Pytest functionality.
- Pytest provides control over test runs and allows test discovery, which is extremely simple with naming conventions.
- Pytest supports data parameterization.

Install Pytest

Install using the `pip install Pytest` command, as shown in *Figure 9.1*:

```
(venv) yogashivamathivanan@Yogashivas-MacBook-Pro Frameworks % pip install pytest
Collecting pytest
  Downloading pytest-7.2.1-py3-none-any.whl (317 kB)
    ██████████ | 317 kB 1.6 MB/s
Collecting pluggy<2.0,>=0.1.2
  Downloading pluggy-1.0.0-py2.py3-none-any.whl (13 kB)
Collecting iniconfig
  Downloading iniconfig-2.0.0-py3-none-any.whl (5.9 kB)
Collecting exceptiongroup>=1.0.0rc8
  Downloading exceptiongroup-1.1.0-py3-none-any.whl (14 kB)
Collecting attrs>=19.2.8
  Downloading attrs-22.2.0-py3-none-any.whl (49 kB)
    ██████████ | 49 kB 15.2 MB/s
Collecting packaging
  Downloading packaging-23.0-py3-none-any.whl (42 kB)
    ██████████ | 42 kB 4.5 MB/s
Collecting tomli>=1.0.0
  Downloading tomli-2.0.1-py3-none-any.whl (12 kB)
Installing collected packages: tomli, pluggy, packaging, iniconfig, exceptiongroup, attrs, pytest
Successfully installed attrs-22.2.0 exceptiongroup-1.1.0 iniconfig-2.0.0 packaging-23.0 pluggy-1.0.0 pytest-7.2.1 tomli-2.0.1
```

Figure 9.1: Install Pytest from Command Line

Verify the installation using the `pytest --version` command, as shown in *Figure 9.2*:

```
(venv) yogashivamathivanan@Yogashivas-MacBook-Pro PytestDemo % pytest --version
pytest 7.2.1
(venv) yogashivamathivanan@Yogashivas-MacBook-Pro PytestDemo %
```

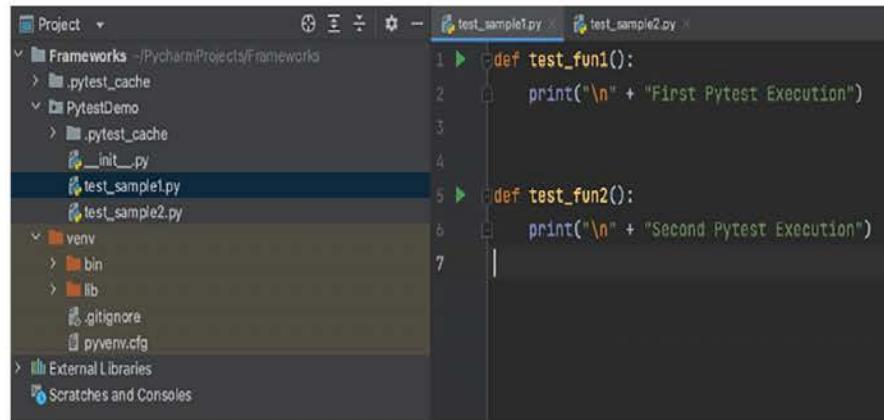
Figure 9.2: Validate Pytest is installed

Writing test with Pytest

In Pytest, there are naming conventions that are required to be followed while naming the test files and functions that help Pytest to identify and run the tests.

By default, Pytest looks for test files with the prefix “`test_`” or suffix “`_test`”. For example, `test_sample.py` or `sample_test.py` respectively. When Pytest executes the test, it searches for the file with the mentioned format in the current directory and subdirectories and identifies all of them as test files. For example, consider the `test/` directory with files `test_sample1.py`, `test_sample2.py`, and a subdirectory with file `test_subsample1.py`. Pytest will consider all the files as test files. We can make Pytest run the other filenames by explicitly mentioning them. Similarly, Pytest looks for functions with the prefix “`test_`”; for example, “`def test_fun1()`”. These naming conventions in Pytest make it easy to identify and run tests, and also make the code more readable. The test name and file name naming convention are shown in *Figure 9.3*.

9.3:



The screenshot shows the PyCharm interface. On the left is the Project tool window with a tree view of files. It includes a 'Frameworks' section with 'pytest_cache', a 'PytestDemo' section with 'init.py', 'test_sample1.py', and 'test_sample2.py', and a 'venv' section with 'bin' and 'lib'. Below these are 'External Libraries' and 'Scratches and Consoles'. The right side of the interface shows two tabs: 'test_sample1.py' and 'test_sample2.py'. The code in 'test_sample1.py' is:

```
def test_fun1():
    print("\n" + "First Pytest Execution")
def test_fun2():
    print("\n" + "Second Pytest Execution")
```

Figure 9.3: Naming convention in Pytest

210 ■ Selenium and Appium with Python

Executing Pytest from PyCharm and Command Line

We can run the test directly by editing and adding configuration in PyCharm, as shown in *Figure 9.4* and *Figure 9.5*:

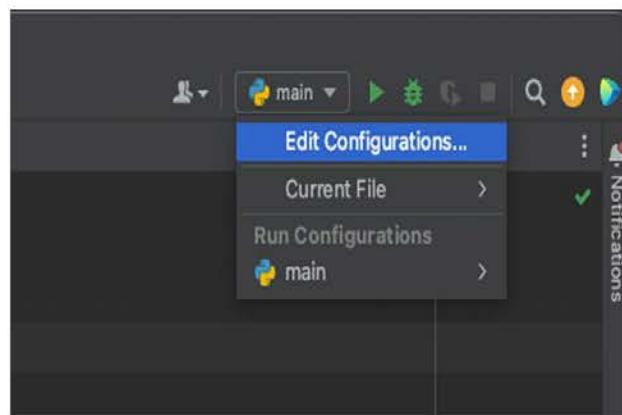


Figure 9.4: Edit configuration in PyCharm -1

Refer to *Figure 9.5* as well:



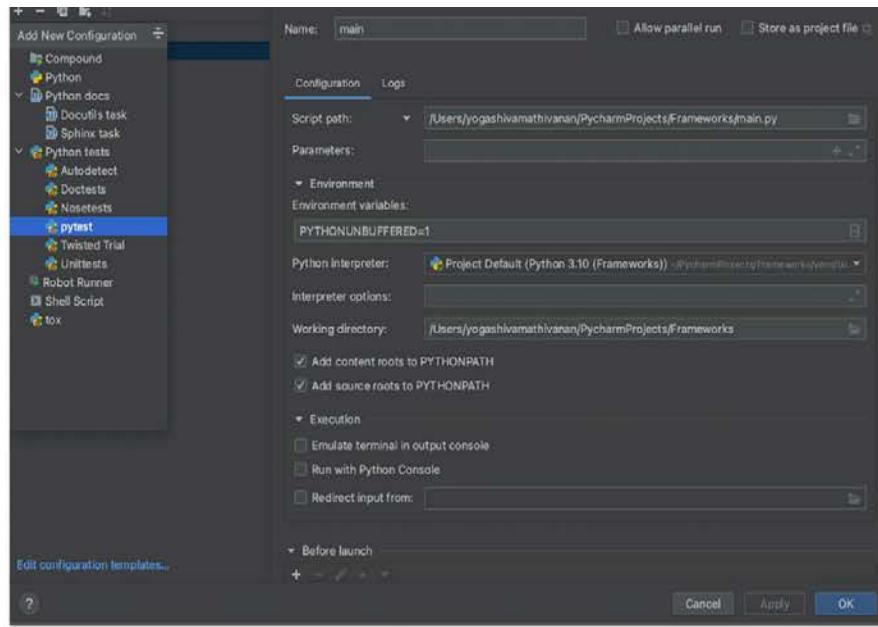


Figure 9.5: Edit configuration in PyCharm -2

After adding the configuration, click on the **Run** button and the test will be executed, as shown in *Figure 9.6*:

```
(venv) yogashivamathivanan@yogashivamathivanan-MacBook-Pro: Frameworks % pytest -v -s
=====
test session starts =====
platform darwin -- Python 3.10.7, pytest-7.2.1, pluggy-1.0.0 -- /Users/yogashivamathivanan/PycharmProjects/Frameworks/venv/bin/python
cachedir: .pytest_cache
rootdir: /Users/yogashivamathivanan/PycharmProjects/Frameworks
collected 3 items

PytestClass/test_sample1.py::test_fun1
First Pytest Execution
PASSED
PytestClass/test_sample1.py::test_fun2
Second Pytest Execution
PASSED
PytestClass/test_sample1.py::test_fun3
Third Pytest Execution
FAILED
=====
        def test_fun3():
            print("In" + "Third Pytest Execution")
            assert 1 == 2, "One is not equal to Two"
            E: assertionerror: One is not equal to Two
            assert 1 == 2

PytestClass/test_sample2.py::AssertionError
=====
short test summary info
FAILED PytestClass/test_sample2.py::test_fun3 - Assertionerror: One is not equal to Two
=====
1 failing, 2 passed in 0.03s
```

```
(venv) yogashivamathivanan@Yogashivas-MacBook-Pro PytestDemo %
```

Figure 9.6: Executing test in PyCharm

Alternatively, the test can be executed using the terminal navigating to the project folder location with all the tests, and using the command **pytest** or **py.test**, as shown in *Figure 9.7*. Each of the functions is considered as an individual test. **Fun3** has assertion failure and **pytest** is failing the result accurately.

In the command “**pytest -v -s**”, the option ‘-v’ is used to increase the verbosity of the output result and option ‘-s’ is used to print **stdout** and **stderr** output from the test to the console, as shown in *Figure 9.7*:

```
PytestDemo -- zsh - 139x42
(venv) yogashivamathivanan@Yogashivas-MacBook-Pro PytestDemo % pytest -v -s
=====
platform darwin -- Python 3.10.7, pytest-7.2.1, pluggy-1.0.0 -- /Users/yogashivamathivanan/PycharmProjects/Frameworks/venv/bin/python
cachedir: .pytest_cache
rootdir: /Users/yogashivamathivanan/PycharmProjects/Frameworks/PytestDemo
collected 3 items

test_sample1.py::test_fun1
First Pytest Execution
PASSED
test_sample1.py::test_fun2
Second Pytest Execution
PASSED
test_sample2.py::test_fun3
Third Pytest Execution
FAILED

=====
                         FAILURES
=====
                         test_fun3

def test_fun3():
    print("\n" + "Third Pytest Execution")
>     assert 1 == 2, "One is not equal to Two"
E     AssertionError: One is not equal to Two
E     assert 1 == 2

test_sample2.py:3: AssertionError
=====
                         short test summary info
=====
FAILED test_sample2.py::test_fun3 - AssertionError: One is not equal to Two
=====
                         1 failed, 2 passed in 0.03s
(venv) yogashivamathivanan@Yogashivas-MacBook-Pro PytestDemo %
```

Figure 9.7: Executing test in the Terminal Using the command in Pytest

A specific test file can be executed by explicitly adding the file name within the execution command like this: “**pytest test_sample1.py -v -s**” as shown in *Figure*

9.8.

```
(venv) yogashivanathivanan@YogaShivas-MacBook-Pro pytestDemo % pytest test_sample1.py -v
=====
platform darwin -- Python 3.10.7, pytest-7.2.1, pluggy-1.0.0 -- /Users/yogashivanathivanan/PycharmProjects/Frameworks/venv/bin/python
cachedir: .pytest_cache
rootdir: /Users/yogashivanathivanan/PycharmProjects/Frameworks/PytestDemo
collected 2 items

test_sample1.py::test_func1
First Pytest Execution
PASSED
test_sample1.py::test_func2
Second Pytest Execution
PASSED

===== 2 passed in 0.06s =====

(venv) yogashivanathivanan@YogaShivas-MacBook-Pro pytestDemo % pytest test_sample2.py -v
=====
platform darwin -- Python 3.10.7, pytest-7.2.1, pluggy-1.0.0 -- /Users/yogashivanathivanan/PycharmProjects/Frameworks/venv/bin/python
cachedir: .pytest_cache
rootdir: /Users/yogashivanathivanan/PycharmProjects/Frameworks/PytestDemo
collected 1 item

test_sample2.py::test_func3
Third Pytest Execution
FAILURE

===== FAILURES =====
test_func3

def test_func3():
    print("In" + "Third Pytest Execution")
    assert 1 == 2, "One is not equal to Two"
    assert 1 == 2, "AssertionError: One is not equal to Two"
    assert 1 == 2

test_sample2.py:3: AssertionError
short test summary info
FAILURE test_sample2.py::test_func3 - AssertionError: One is not equal to Two
===== 1 failed in 0.01s =====

(venv) yogashivanathivanan@YogaShivas-MacBook-Pro pytestDemo %
```

Figure 9.8: Execute specific files using the command line in Pytest

Moreover, to run specific functions or test with Pytest, we can explicitly mention the text contained in the function name. Let us say multiple functions has text “payment” within the function name, such as “**test_applePayPayment():**” or “**test_creditCardPayment():**”. In this case, these specific functions or tests can be executed using the command line option “**-k**” that takes the substring expression from function name for test execution, as “**pytest -k Payment -v -s**”. This can be seen in *Figure 9.9*:

```
(venv) yogashivanathivanan@YogaShivas-MacBook-Pro pytestDemo % pytest -k Payment -v -s
=====
platform darwin -- Python 3.10.7, pytest-7.2.1, pluggy-1.0.0 -- /Users/yogashivanathivanan/PycharmProjects/Frameworks/venv/bin/python
cachedir: .pytest_cache
rootdir: /Users/yogashivanathivanan/PycharmProjects/Frameworks/PytestDemo
collected 2 items / 1 deselected / 2 selected

test_sample1.py::test_funcPayment
First Pytest Execution
PASSED
test_sample2.py::test_funcPayment
Third Pytest Execution
FAILURE

===== FAILURES =====
test_funcPayment

def test_funcPayment():
    print("In" + "Third Pytest Execution")
    assert 1 == 2, "One is not equal to Two"
    assert 1 == 2, "AssertionError: One is not equal to Two"
    assert 1 == 1

test_sample2.py:3: AssertionError
short test summary info
FAILURE test_sample2.py::test_funcPayment - AssertionError: One is not equal to Two
===== 1 failed in 0.01s =====

(venv) yogashivanathivanan@YogaShivas-MacBook-Pro pytestDemo %
```



```
===[Terminal Window]=====
short test summary info
FAILED test_sample2.py::test_funPayment - AssertionError: One is not equal to Two
=====
1 failed, 1 passed, 1 reselected in 0.04s
=====
(venv) yogeshivamchavhan/Pyogeshi/venv-MacBook-Pro:Pyogeshi
```

Figure 9.9: Execute the test using regular expression in Pytest

Markers in Pytest

Command line option ‘-m’ is used to execute the tests marked with a certain marker. For example, `pytest -m smoke` will run all the tests that are marked ‘smoke’. When we run the command “`pytest -m smoke -k Payment -v -s`”, there will be a potential warning like this: “`PytestUnknownMarkWarning: Unknown pytest.mark. smoke - is this a typo?`”

You can register custom marks to avoid this warning. For details, see <https://docs.pytest.org/en/stable/how-to/mark.html>

Refer to the following code “`test_sample1.py`” to see how to use markers in pytest:

CODE – `test_sample1.py`

1. `import pytest`
- 2.
- 3.
4. `@pytest.mark.smoke`

```

5. def test_fun1Payment():
6.     print("\n" + "First Pytest Execution")
7.
8. @pytest.mark.regression
9. def test_fun2():
10.    print("\n" + "Second Pytest Execution")

```

This warning is because pytest is not recognizing the marker `smoke`. To avoid the warning, create a file `pytest.ini` in the root directory and add intended markers as in the following code, “CODE – `pytest.ini`”. We can also find out the list of markers in pytest, and both default and added markers will be displayed using the command “`pytest --markers`”.

CODE – `pytest.ini`

```

1. [pytest]
2.
3. markers =
4.     smoke: smoke testing related
5.     regression: regression testing related

```

Once the custom markers are added, only the marked test will be executed, as shown in *Figure 9.10*:

```

(yenv) yogansivamathiyanan@Yogash-Virtual-MacBook-Pro:~/PycharmProjects/PytestDemo% pytest -n smoke -k Payment -v -s
=====
platform darwin -- Python 3.10.7, pytest-7.2.1, pluggy-1.0.0 -- /Users/yogansivamathiyanan/PycharmProjects/Frameworks/PytestDemo/vm/bin/python
cachedir: .pytest_cache
rootdir: /Users/yogansivamathiyanan/PycharmProjects/Frameworks/PytestDemo, configfile: pytest.ini
collected 4 items / 3 deselected / 1 selected

Test_sample1.py::test_fun1Payment
First Pytest Execution
PASSED

=====
1 passed, 3 deselected in 0.01s

```

Figure 9.10: Executing test using Marker in Pytest

We can also skip the test by using “`not`” in the command, as “`pytest -m "not smoke" -k "not Payment" -v -s`” and also by using marker skip using annotation `@pytest.mark.skip` and then execute all the tests using command “`pytest -v -s`”. Then the test with marker skip will be skipped with a message in the output, as “`test_sample1.py::test_fun1Payment SKIPPED (unconditional skip)`”.

Additionally, using marker annotation `xfail` to the function like “`@pytest.mark.xfail`”, pytest will execute the test but will not show the result in the final report. In the output, it will show the test as `xpassed`, as shown in *Figure 9.11*:


```
(venv) yogashivanathvaran@Yogashivas-MacBook-Pro pytestDemo % pytest -m smoke -v -s
=====
platform darwin -- Python 3.10.7, pytest-7.2.1, pluggy-1.8.0 -- /Users/yogashivanathvaran/PycharmProjects/Frameworks/venv/bin/python
--omicsrc: .pytest_cache
rootdir: /Users/yogashivanathvaran/PycharmProjects/Frameworks/PyTestDemo, configfile: pytest.ini
collected 4 items / 3 deselected / 1 selected

test_sample2.py::test_fn1<skip>
Fourth Pytest Execution
=====

(venv) yogashivanathvaran@Yogashivas-MacBook-Pro pytestDemo %
```

Figure 9.11: Skipping the test using Marker in Pytest

Fixtures in Pytest

In Pytest, a fixture is a function that provides a fixed baseline for a test. It is essentially a piece of reusable code that is executed before and / or after the test, like set up and tear down functions. A fixture can provide a specific context, data, setup resources, or environment for the test to run in. It can also be used to avoid writing the same setup code multiple times for different tests. To use a fixture in a test function, you simply pass the fixture function as an argument to the test function.

In the following CODE 1, the function “`fixtureSetup`” has the annotation `@pytest.fixture` to indicate the function is a fixture. Using the random module, the function returns a random integer between 1 to 100. The function `fixtureDemo` has the annotation `@pytest.mark.fixtureDemo` and takes the `fixtureSetup` fixture as an argument. `fixtureDemo` function validates that the returned number is an integer and within the 1 to 100 range, as expected using an assert statement. In the output, when `fixtureDemo` is executed using the command `pytest -m fixtureDemo -v -s`, the fixture `fixtureSetup`function is automatically invoked and executed before the `fixtureDemo` function.

CODE 1 – test_sample1.py

```
1. import pytest
2. import random
3.
4. @pytest.fixture
5. def fixtureSetup():
6.     print("\n" + "Pytest Fixture to Setup")
7.     return random.randint(1, 100)
8.
9.
```



```
10. @pytest.mark.fixtureDemo  
11. def test_fixtureDemo(fixtureSetup):  
12.     print("\n" + "Pytest Demo Flow Execution")  
13.     assert isinstance(fixtureSetup, int)  
14.     assert 1 <= fixtureSetup <= 100
```

Moreover, tear-down instructions can be added in the same fixture function by separating the code using the “yield” statement as in the following CODE 2. The output will have a teardown statement at the end of execution if “yield” is present in the fixture.

In real-time, fixture can be used to set up driver instance or invoke the browser, or some configuration properties at the beginning of execution and can be used to tear down the driver, clear cookies, and so on, at the end of the execution.

CODE 2–test_sample1.py

```
1. @pytest.fixture  
2. def fixtureSetup():  
3.     print("\n" + "Pytest Fixture to Setup")  
4.     yield  
5.     print("\n" + "Pytest Fixture to TearDown")
```

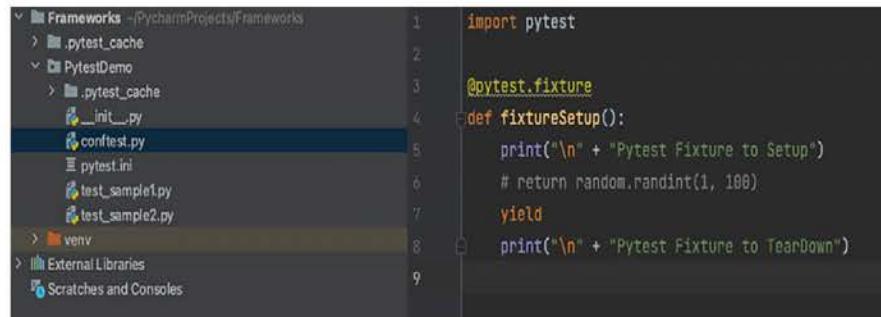
OUTPUT: pytest -m fixtureDemo -v -s

```
1. test_sample1.py::test_fixtureDemo  
2. Pytest Fixture to Setup  
3.  
4. Pytest Demo Flow Execution  
5. PASSED  
6.  
7. If Yield Statement Present in fixture  
8. Pytest Fixture to TearDown
```

Reusing Fixtures in Pytest

The fixtures written in a single test file in pytest are only available to functions present in the same file. However, to have the fixture accessibility across all the files present in a directory, we would need to create a file named “**conftest.py**” in the same directory, as shown in *Figure 9.12*. **conftest.py** is a special Python module used by

pytest to define fixtures, plugins, and other configurations that can be shared across multiple test modules.



The screenshot shows a PyCharm interface with a file tree on the left and a code editor on the right. The file tree includes a project named 'Frameworks' containing '.pytest_cache', 'PytestDemo' (with files '_init_.py', 'conftest.py', 'pytest.ini', 'test_sample1.py', 'test_sample2.py'), 'venv', 'External Libraries', and 'Scratches and Consoles'. The code editor shows a Python file with the following content:

```

1 import pytest
2
3 @pytest.fixture
4 def fixtureSetup():
5     print("\n" + "Pytest Fixture to Setup")
6     # return random.randint(1, 100)
7     yield
8     print("\n" + "Pytest Fixture to TearDown")

```

Figure 9.12: Fixtures in Pytest

The fixture in `conftest.py` can be reused at the class level by using annotation `@pytest.mark.usefixtures("fixtureSetup")` as in the following code. As in the output, fixture code before `yield` is executed before every test, followed by a teardown statement after the `yield` statement, as shown in the preceding Figure 9.12.

However, if the fixture is returning any data and the use case is to use the returned data in a test, then we need to pass the fixture as an argument to that function, as in line 15 in the following CODE 1.

In pytest, the “scope” parameter in fixtures, allows control over when the fixture is set up and tear down. There are four possible values for the scope parameter:

- **function (default):** The fixture is set up and torn down once per test.
- **class:** The fixture is set up once per test class and torn down once after all test methods in the class have been completed.
- **module:** The fixture is set up once per test module and torn down once after all test methods in the module have been completed.
- **session:** The fixture is set up once per test session and tear down once after the end of the test session.

As in CODE 2, if we add the scope as a class, then the setup statement before “`yield`” is executed once before the class. The teardown statement after “`yield`” is executed once after all the methods in the class have been completed, as seen in OUTPUT 2.

CODE 1 – test_sample3.py

1. `import pytest`
- 2.
3. `@pytest.mark.usefixtures("fixtureSetup")`
4. `class Test_Sample3:`

```
5.  
6.     @pytest.mark.fixtureDemo  
7.     def test_fun5(self):  
8.         print("\n" + "This is Function 5 in the Class Sample 3")  
9.  
10.    @pytest.mark.fixtureDemo  
11.    def test_fun6(self):  
12.        print("\n" + "This is Function 6 in the Class Sample 3")  
13.  
14.    @pytest.mark.fixtureDemo  
15.    def test_fun6(self, fixtureSetup):  
16.        print("\n" + "This is Function 6 in the Class Sample 3" +  
             str(fixtureSetup))
```

CODE 2 – conftest.py

```
1. @pytest.fixture(scope="class")  
2. def fixtureSetup():
```

OUTPUT 1

```
1. collected 2 items  
2. test_sample3.py::Test_Sample3::test_fun5  
3. Pytest Fixture to Setup  
4.  
5. This is Function 5 in the Class Sample 3  
6. PASSED  
7. Pytest Fixture to TearDown  
8.  
9. test_sample3.py::Test_Sample3::test_fun6  
10. Pytest Fixture to Setup  
11.  
12. This is Function 6 in the Class Sample 3  
13. PASSED  
14. Pytest Fixture to TearDown
```

OUTPUT 2

1. collected 2 items

2. test_sample3.py::Test_Sample3::test_fun5
3. Pytest Fixture to Setup
- 4.
5. This is Function 5 in the Class Sample 3
6. PASSED
7. test_sample3.py::Test_Sample3::test_fun6
8. This is Function 6 in the Class Sample 3
9. PASSED
10. Pytest Fixture to TearDown

Parameterization in fixtures in Pytest

Parameterization in fixtures allows the creation of a fixture with one or more arguments, which can be a value or set of values. This is beneficial when fixtures need to be initialized with multiple values, for example, when testing a function with multiple input values.

In the following example, the `loginfixture` fixture is parameterized with a list of tuples, where each tuple contains a set of a username and a password. The `request` object is used to access the parameter value for each test run. The `test_fun6` function uses the `loginfixture` fixture as an argument. Because the `loginfixture` fixture is parameterized with a list of three different sets of values, the `test_fun6` test will be run three times, once for each set of values in the `loginfixture`. When the test file is run with the command `pytest test_sample4.py -v -s`, pytest will run the `test_fun6` function/test three times and print three sets of username and password.

CODE 1 – test_sample4.py

```
1. def test_fun6(loginfixture):
2.     print("\n" + f"This is username {loginfixture[0]} and password
is {loginfixture[1]}")
```

CODE 2 – conftest.py

```
1. import pytest
2.
```

```
3. @pytest.fixture(params=[("username1", "password1"), ("username2",
   "password2"), ("username3","password3")])
4. def loginfixture(request):
5.     return request.param
```

OUTPUT

```
1. collected 3 items
2.
3. test_sample4.py::test_fun6[loginfixture0]
4. This is username username1 and password is password1
5. PASSED
6. test_sample4.py::test_fun6[loginfixture1]
7. This is username username2 and password is password2
8. PASSED
9. test_sample4.py::test_fun6[loginfixture2]
10. This is username username3 and password is password3
11. PASSED
```

Soft Assert in pytest – Softest

As discussed, the assertion using Python is used to check the expected and actual values of the test case. If the condition in the assert statement is ‘False’, then the test case will fail, and the execution of the test case will stop. The package “softest” allows to perform soft assertions in the test cases, which are validation points, but they do not stop the execution of the test case if a condition is ‘False’. Instead, they continue the test execution, and the failed assertions are recorded as non-blocking error for later reporting.

In the following code, the ‘**TestExample**’ class extends the ‘**TestCase**’ class from Softest and in the test, we can use **self.soft_assert** method to perform soft assertion using the **self.assertEqual** assertion method. After the test, the **self.assert_all()** method is called; it checks if there were any soft assertion failures and raises an assertion error if any. If there were no failures, the test case passes. As in the output, all the tests were executed even with the failures and the tests continued the execution and failures are recorded at the end.

CODE 1 – test_sample5.py and test_sample6.py (Same Code)

```
1. import pytest
2. from softest import TestCase
3.
4.
5. class TestExample2(TestCase):
6.     def test_addition2(self):
7.         a = 2
8.         b = 2
9.         result = a + b
10.        self.soft_assert(self.assertEqual, result, 5)
11.        if result == 5:
12.            print("PASSED")
13.        else:
14.            print("FAILED")
15.        print("assert 1 Passed++++++")
16.
17.        a = 3
18.        b = 3
19.        result = a + b
20.        self.soft_assert(self.assertEqual, result, 7)
21.        if result == 7:
22.            print("PASSED")
23.        else:
24.            print("FAILED")
25.        print("assert 2 Passed++++++")
26.
27.        self.assert_all()
```

OUTPUT

The output can be seen in the following *Figure 9.13*:

```
(venv) yogeshivamathivanan@yogeshivamathivanan-MacBook-Pro:~/PytestDemo$ pytest test_softassert.py -v
=====
platform darwin -- Python 3.10.7, pytest-7.2.1, pluggy-1.0.0 -- /Users/yogeshivamathivanan/PycharmProjects//frameworks/venv/bin/python
cachedir: .pytest_cache
duration: 3.10.7, Platform: macos-12.6-x86_64-64bit, Packages: {'pytest': '7.2.1', 'pluggy': '1.0.0'}, Plugins: {'allure-pytest': '2.18.1', 'html': '3.2.0', 'netdata': '2.0.4'}, _SAVING_HOME: '/Library/Java/JavaVirtualMachines/jdk-17.0.4.1.jdk/Contents/Home'
rootdir: /Users/yogeshivamathivanan/PycharmProjects//frameworks/PytestDemo, configfile: pytest.ini
plugins: allure-pytest-2.18.1, html-3.2.0, metadata-2.0.4
collected 2 items

test_softassert/test_sample5.py::TestExample::test_addition FAILED
assert 3 Passed=====
assert 4 Failed=====
assert 4 Passed=====
assert 5 Failed=====
test_softassert/test_sample6.py::TestExample2::test_addition2 FAILED
assert 1 Passed=====
assert 2 Failed=====
assert 2 Passed=====
assert 3 Failed=====

=====
===== FAILURES =====
===== testsoftassert.TestExample::test_addition =====
E: assert 3 Failed
E: assert 4 Failed
E: assert 5 Failed
=====
===== abort test summary info =====
FAILS test_softassert/test_sample5.py::TestExample::test_addition - AssertionError: === soft assert failure details follow below ===
FAILS test_softassert/test_sample6.py::TestExample2::test_addition2 - AssertionError: === soft assert failure details follow below ===
2 failed in 01.00s
```

Figure 9.13: Output

HTML reports in Pytest using pytest-html

HTML reports can be generated for the test runs in Pytest using the plugin **pytest-html**. The generated html reports can be viewed in the browser. Pytest-html plugin provides an easy way to view test run results. Its customization and integration with Pytest makes it a powerful tool for automation. Here are some features of the **pytest-html**:

- **Test Execution Summary:** The html report provides a complete summary of the tests with test status, including passed, failed, skipped, xpassed, and so on.
- **Screenshots:** This is the most powerful feature of the **pytest-html** report; it can capture screenshots during the test runs and include them in the html report.
- **Test details:** The report provides detailed information on each test with status, duration, failure message, screenshots, or traceback information.

- **Logs:** The report displays log output from the test.
- **Customization:** The report title, report file name, and so on, can be customized for the report output.

- **Organized test hierarchy:** If the test hierarchy is organized, then the html report will display the hierarchy of the test suite.

pytest-html Install

Install the plugin using pip: `pip install pytest-html`

Generate pytest-html report

Run the test with the option “`--html=report.html`”. The option tells Pytest to generate an HTML report and save it to a file with the name `report.html`. For example, running the command “`pytest -v -s --html=report.html`” will generate the report in the project directory.

View the report

The report can be opened in any browser and will have test run details, as shown in *Figure 9.14*:

report.html

Report generated on 16-Feb-2023 at 17:24:07 by `pytest-html` v3.2.0

Environment

JAVA_HOME	/Library/Java/JavaVirtualMachines/jdk-17.0.4.1.jdk/Contents/Home
Plugins	[“pygments”: “11.0.0”, “pytest”: “7.2.1”]
Platform	macOS-12.6-x64_54-086-64bit
Plugins	[“html”: “3.2.0”, “matacdata”: “2.0.4”]
Python	3.10.7

Summary

10 tests ran in 0.04 seconds.

(Un)check the boxes to filter the results.

10 passed, 0 skipped, 0 failed, 0 errors, 0 expected failures, 0 unexpected passes

Results

Show all details / Hide all details

Result	Test	Duration	Links
Passed	test_sample1.py::test_AutIPayment	0.00	
Passed	test_sample1.py::test_AutI2	0.00	
Passed	test_sample1.py::test_ExureDemo	0.00	
Passed	test_sample2.py::test_AutIPayment	0.00	
Passed	test_sample2.py::test_AutIShipping	0.00	
Passed	test_sample3.py::Test_Sample0::test_Iun6	0.00	
Passed	test_sample3.py::Test_Sample0::test_Iun6	0.00	
Passed	test_sample4.py::test_AutIgInitiate0	0.00	
Passed	test_sample4.py::test_AutIgInitiate1	0.00	
Passed	test_sample4.py::test_Iun0@grInitiate0	0.00	

Figure 9.14: Generating html report in Pytest

Advantages of Pytest

Pytest offers several advantages, including:

- Pytest is free and open source, which makes it accessible to developers of all backgrounds and budgets.
- Pytest can run multiple tests in parallel in less time.
- Pytest can detect test files and functions and saves time and effort.
- Pytest can skip a subset of tests during execution, to run specific tests or sections of code and is useful for debugging or troubleshooting.
- Pytest has a simple syntax, which makes it easy for developers to get started and write tests quickly and efficiently.

Robot framework

Let us look at some of the other major frameworks. The open-source robot framework is getting popular for automation testing. The robot framework is well suited for acceptance testing, **Acceptance Test Driven Development (ATDD)**, **Behavior Driven Development (BDD)**, and **Robotic Process Automation (RPA)**. It is designed to write test cases in a keyword-driven and simple testing format/approach, to have a readable clear understanding of the test case functionality. Hence, not much programming is required. It is backed and written by Python but supports other programming languages. The robot framework provides test libraries to implement test automation flow and allows user-created custom keywords. The Robot Framework is a flexible and powerful test automation framework that can be integrated with a wide range of tools and technologies. It supports various interfaces and protocols, making it possible to automate different types of systems and applications. It can also be used in distributed and heterogeneous environments, allowing for greater scalability and flexibility in test automation. Overall, the Robot Framework is a versatile and robust solution for automation that can meet the needs of many different organizations and industries.

Why Robot framework?

The various reasons why you should use Robot framework are as follows:

- Robot framework provides a keywords library to fulfill all the automation requirements; for example, a Selenium keywords library to utilize selenium for UI automation.

- Robot Framework is platform-independent, which means that it can run on different operating systems and supports multiple programming languages, such as Python, Java, and so on, making it accessible to a wider range of users.
- Robot framework provides convenient self-generated execution logs and reports, with features such as detailed step information, easier screenshots of steps and auto screenshots of failure steps, and so on.
- Robot framework provides a command line interface and XML-based execution report files for integration into existing builds within continuous integration systems.
- Robot framework has keywords to support testing of web applications, Rest API, mobile applications, database connection, and so on within a single framework.
- Robot framework provides test-case and test-suite level setup and teardown.

High-level architecture

The robot framework is a modular and extensible framework, that allows easier integration with other systems, easier maintenance, and scalability. The robot framework architecture flow is depicted in the following *Figure 9.15*:

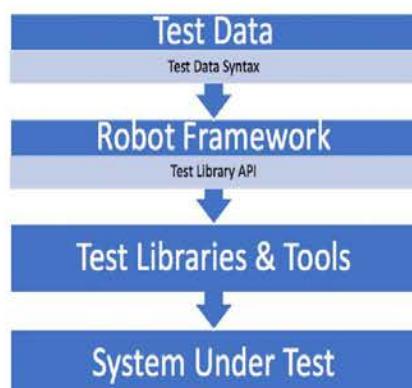


Figure 9.15: Robot Framework architecture

The test data contains the input data for test case execution. The test data format can be of type XML, JSON, CSV, and so on.

The test library is the building block of the robot framework. They provide keywords and functionalities to be used in the automation and interaction with the system under test. Libraries can use the application interface directly or use the tools as drivers.

During the execution process, the robot framework processes the test data, executes test cases using test libraries, and generates test execution reports, and logs.

Robot framework installation

The Robot framework requires Python 3.6 or a newer version to be installed. To use it with Python 2, you can use Robot Framework 4.1.3. Use the pip command “**pip install robotframework**” to install the latest version of the robot framework. Use the commands **pip install robotframework==4.1.3** and **pip install -upgrade robotframework** to install a specific robot version and to upgrade to the latest version of the robot framework respectively.

Use the pip command **pip show robotframework** to verify the version and have a successful installation. The installation and validation of the robot framework are shown in the following *Figure 9.16*:

```
(venv) yogashivamathivanan@Yogashivas-MacBook-Pro PytestDemo % pip install robotframework
Collecting robotframework
  Downloading robotframework-6.0.2-py3-none-any.whl (658 kB)
    ━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 658.7/658.7 kB 1.6 MB/s eta 0:00:00
Installing collected packages: robotframework
Successfully installed robotframework-6.0.2

[notice] A new release of pip is available: 23.0 → 23.0.1
[notice] To update, run: pip install --upgrade pip
(venv) yogashivamathivanan@Yogashivas-MacBook-Pro PytestDemo % pip show robotframework
Name: robotframework
Version: 6.0.2
Summary: Generic automation framework for acceptance testing and robotic process automation (RPA)
Home-page: https://robotframework.org
Author: Pekka Klärck
Author-email: peke@eliga.fi
License: Apache License 2.0
Location: /Users/yogashivamathivanan/PycharmProjects/Frameworks/venv/lib/python3.10/site-packages
Requires:
```

Required-by:

Figure 9.16: Install robot framework

Use the command `pip list` or navigate to Python interceptor in PyCharm preference/settings, to verify that the robot framework is installed in the desired virtual environment, as shown in *Figure 9.17*:

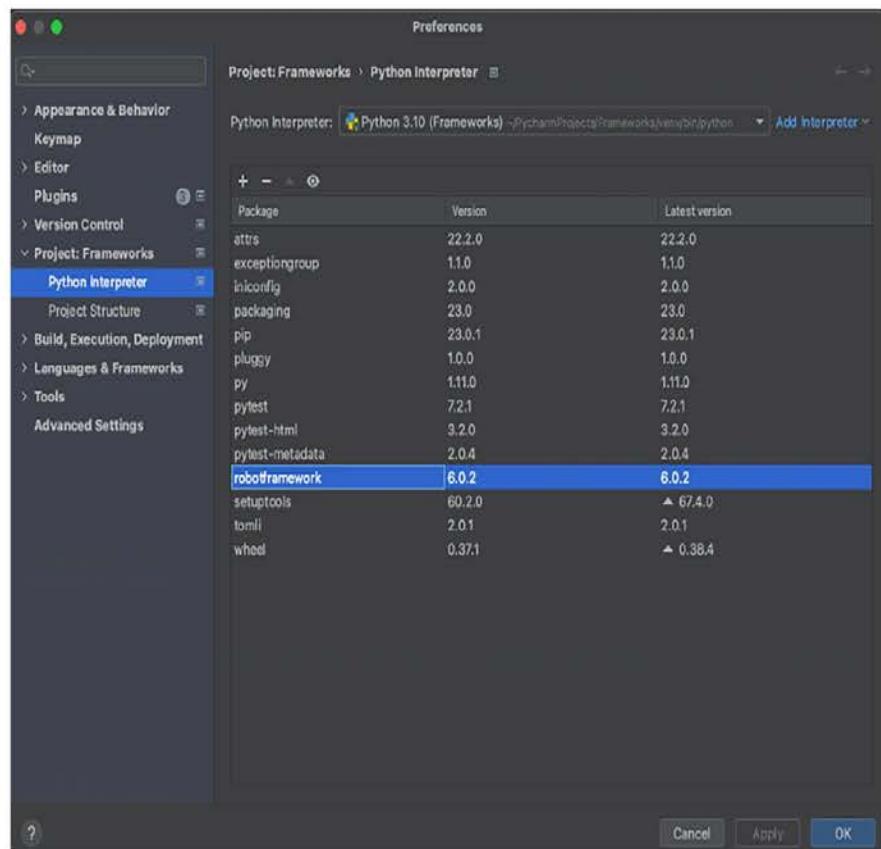


Figure 9.17: Verify the robot framework installed in PyCharm

Robot Framework Libraries

As discussed earlier, the building block of the robot framework is the libraries. The robot framework supports several libraries for different purposes, such as web testing, API testing, database testing, mobile testing, and so on. The list of all the available libraries can be found at <https://robotframework.org/#libraries>. For testing web applications, `robotframework-seleniumlibrary` is required to be

~~For testing web applications, Robot Framework-SeleniumLibrary is required to be installed using the command `pip install robotframework-seleniumlibrary`.~~

Let us understand in-depth how to use keywords in the Selenium library within the robot framework. Navigate to the documentation of the robot framework selenium library using the link <https://robotframework.org/SeleniumLibrary/SeleniumLibrary.html>. The SeleniumLibrary uses the Selenium WebDriver internally to control the web browsers. There are keywords to work with all the UI elements using different locator strategies such as ID, Xpath, CSS, and so on, as in the following *Table 9.1*:

Strategy	Match based on	Example
id	Element id.	id:example
name	name attribute.	name:example
identifier	Either id or name.	identifier:example
class	Element class.	class:example
tag	Tag name.	tag:div
xpath	XPath expression.	xpath://div[@id="example"]
css	CSS selector.	css:div#example
dom	DOM expression.	dom:document.images[5]
link	Exact text a link has.	link:The example
partial link	Partial link text.	partial link:he ex
data	Element data-* attribute	data:id:my_id
jquery	jQuery expression.	jquery:div.example
default	Keyword specific default behavior.	default:example

Table 9.1: Robot Framework Locators

To click an element, use the keyword “**Click Element**” with one of the locator strategies to identify the element. Most importantly, there should be at least a tab space between any keyword and argument. The format of the locator strategy is either **strategy:value** or **strategy=value**, or when using id, we can directly mention the value. Here are a few examples:

Click Element	id:submit
Click Button	xpath://button[@id='submin']
Input Text	id=username admin
Input Text	username admin

Robot Framework keywords

Once the required libraries are installed, let us look at different keywords allowed in the robot framework and how to create an ideal folder structure to add our test cases and syntaxes used in the robot framework.

There are several keywords in the robot framework:

- **Built-in Keywords:** These are the pre-defined robot framework-provided keywords to identify different sections and functionalities, such as setting up the test environment, identifying configuration, test cases, variables, and so on. Here are some of the commonly used built-in keywords in the robot framework:

- o **Settings:** The keyword is used to configure the behavior of the robot framework. Variables, test data, and library locations are defined within the settings keyword.
- o **Variables:** The values to be used in the test execution are defined within the variables keyword.
- o **Test Cases:** The test steps are within the test cases keyword. The test steps are keywords (either user-defined or library).
- o **Keywords:** These are the actions performed in a test case.
- **User-defined Keywords:** These are the keywords defined by the users either by using a combination of built-in keywords or by writing Python functions.
- **External Library Keywords:** The keywords are provided by the installed libraries to perform specific desired actions.

Robot Framework Folder Structure

The folder structure in the Robot framework is a key step in making the test automation project more robust, organized, and easy to manage. The main folders and files added in the robot framework within the root folder are test suites, test cases, test data, resources, config, and library. The test files have the extension “**.robot**” and the resources file has the extension “**.robot**” or “**.resource**”.

The folders and files are explained as follows, and shown in *Figure 9.17*:

- **Test Suite folder:** The test suite folder contains all the test suite files and contains the test cases to be executed. The test cases contain keywords. When the complete Test suite is executed, all the test cases within the suite are executed.
- **Resource Folder:** The resource folder is where both user-defined and the library reusable keywords are defined. These keywords can be shared across multiple test suites and test cases. To use the keywords in resource files, these files are required to be imported into the test suite using the “**Resource**” keyword.
- **Test Data Folder:** Test Data folder is used to store all the data files required for the automation of the project. It can be API request payload data to access APIs, database query data, or excel sheet data for data parameterization.
- **Configs and Libraries:** The Python code implementation of user-defined keywords are added in the Python file in the library folder, and test execution configurations such as test environments, environment variables from the config file, and so on, are added in the configs folder.

- **Results Folder:** This is the location where test execution results are stored. When the robot framework is executed, the log and report files are stored in the results folder. By default, these files are created in the root directory, but can be configured to be placed in the results folder. In addition to log file, the results folder can contain screenshots generated by the robot framework.

Refer to the following *Figure 9.18*:

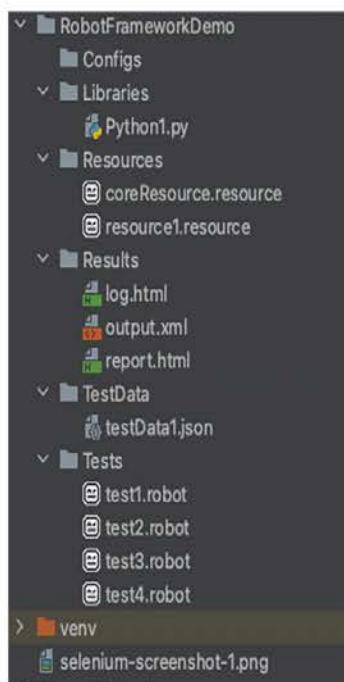


Figure 9.18: Robot Framework folder structure

Folder structure best practice

Some of the best practices are:

- As per project demand, create a separate folder for specific contents of the application for easier managing and maintaining the project.
- Group related test cases and resources in the same folder.
- Follow consistency in naming the project folder and file, for easier understanding of the contents.
- Make sure the contents are not repeated across multiple folder structures.

Test Data Management in Robot Framework

Test data is an essential component of any automation framework. Any project will have to deal with data for parameterization, store collected test data during execution to validate, JSON/XML payload data for APIs, and so on. This data needs to be organized and managed for effective and optimum test automation. In the following code, we are storing the URL, username, and password as variables, that can be referenced across the test suites and imported resources making it easy to update the test data as needed.

CODE – test3.robot

```
1. *** settings ***
2. Library      SeleniumLibrary
3.
4. *** variables ***
5. ${url}          https://magento.softwaretesting-
                    board.com/customer/account/login
6. ${username}    admin12@gmail.com
7. ${password}    admin12!@
8. ${name}        Admin
9.
10.*** Test Cases ***
11.Login Test
12.  Open Browser           browser=chrome
13.  Goto                  ${url}
14.  Input Text             email           ${username}
15.  Input Password         pass            ${password}
16.  Click Element          xpath://*[@id="send2"]
17.  Wait Until Page Contains  Welcome, admin admin!
18.  Close Browser
```


OUTPUT

The output can be seen in the following *Figure 9.19*:

```
Test3
=====
Login Test | PASS |

Test3 | PASS |
1 test, 1 passed, 0 failed
=====
Output: /Users/yogashivamathivanan/PycharmProjects/Frameworks/output.xml
Log:   /Users/yogashivamathivanan/PycharmProjects/Frameworks/log.html
Report: /Users/yogashivamathivanan/PycharmProjects/Frameworks/report.html

Process finished with exit code 0
```

Figure 9.19: Output

Another example is in the following “CODE – test4.robot”, where we are using OperatingSystem Library to read the test data from the testData1.json file, which can be used to request an API, or for other required purposes.

CODE – test3.robot

1. *** Settings ***
2. Library OperatingSystem
- 3.
4. *** Test Cases ***
5. Reading content of test file
6. \${contents} Get File \${EXECDIR}/
 RobotFrameworkDemo/
 TestData/testData1.json
7. Log to Console \${contents}

OUTPUT

The output can be seen in the following *Figure 9.20*:

```
=====
Test4
=====
Reading content of test file
    "firstName": "Don",
    "lastName": "Bosco",
    "age": 35,
    "address": {
        "street": "123 Main St",
        "city": "Santa Ana",
        "state": "CA",
        "zip": "92701"
    }
|
| PASS |
=====
Test4 | PASS |
1 test, 1 passed, 0 failed
=====
Output: /Users/yogashivamathivanan/PycharmProjects/Frameworks/output.xml
Log:   /Users/yogashivamathivanan/PycharmProjects/Frameworks/log.html
Report: /Users/yogashivamathivanan/PycharmProjects/Frameworks/report.html
=====
Process finished with exit code 0
```

Figure 9.20: Output

Robot framework test execution flow

Understanding test execution flow in the robot framework is crucial for writing effective test cases. Let us look at the execution flow and analyze how it is done in the robot framework.

Test setup/Test suite setup

This is the first step of the test execution flow. All the core configurations and initialization are performed in this step, including setting up the environment, setting global variables, importing libraries and resources, and so on. Test Setup is executed once before each test case, and test suite setup is executed once before the test suite, depending on test requirements. The purpose of the test case or test suite setup is to prepare the test case or test suite for the actual testing to begin.

In the following “**CODE 1 - test1.robot**”, we see the test suite file with one test case to validate that the user can log in with valid user credentials. Here, the test setup is to set up the browser. The Open Application is a user-defined keyword and implementation is done in the resources folder in **coreResource.resource**, as in “**CODE 2 - coreResource.resource**”. Moreover, setting up variables, importing resources by providing the resource path as in lines 4 and 5, and importing the external library SeleniumLibrary for UI automation keywords as in line 3, is part of the test setup step.

In another setup step, we can write custom keywords by implementing the logic Python code in a Python file as in **CODE 4 - Python1.py**, where the function **todaysTime()** returns the string with the current date-time and name assigned, while initializing the python1 object. The Python object is initialized in settings in **CODE 1 - test1.robot**, where the library is imported with the argument passed along.

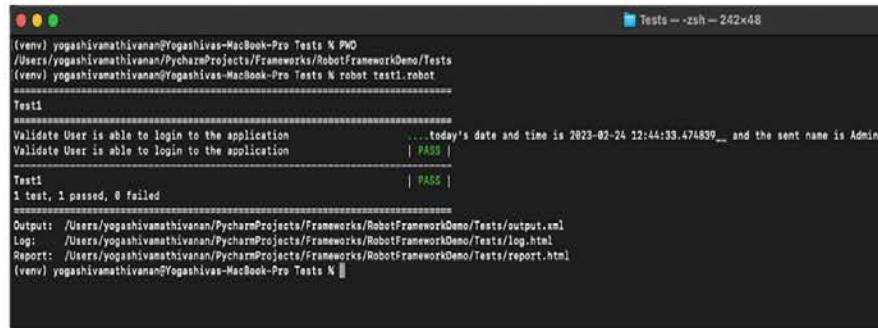
Webdrivermanager

The most important step of UI automation is to install and configure the browser driver. As discussed previously, we can use the **webdrivermanager** package for this. First, install **webdrivermanager** using the pip command “**pip install webdrivermanager**” which will install the browser driver. Now, use the command “**webdrivermanager firefox --linkpath /usr/local/bin**” to move the browser driver to the directory, that contains executable files that is installed locally on the system. The executables installed in this directory can be executed from anywhere in the system. This allows the SeleniumLibrary “**Open Browser**” keyword to open the desired browser without having to provide an **executable_path** argument.

Test case execution

In the following “**CODE 1 - test1.robot**”, the test steps are written in simple English language for all the team members to understand, what the test case is achieving. Robot framework allows the use of Gherkin language, which is in the format of ‘Given’, ‘When’, ‘And’, and ‘Then’ keywords, to make the test step flow clear. This series of test steps or keywords are implemented in **coreResource.resource** and **resource1.resource** as in the following **CODE 2** and **CODE 3**, provided these resource files are imported in the test suite file as in line 4 and line 5 in **CODE 1**. The keywords in this case are implementing the login functionality using SeleniumLibrary which is imported only once in the test suite file.

The test suite can be executed by using the command **robot**, followed by the path of the test suite file. To execute the following **CODE 1**, navigate to the directory containing the test suite file and enter the command **robot test1.robot**, as shown in *Figure 9.21*:



```
(venv) yogashivamathivanan@Yogashivas-MacBook-Pro Tests % PMO
/Users/yogashivamathivanan/PycharmProjects/Frameworks/RobotFrameworkDemo/Tests
(venv) yogashivamathivanan@Yogashivas-MacBook-Pro Tests % robot test1.robot
=====
Test1
=====
Validate User is able to login to the application ...today's date and time is 2023-02-24 12:44:33.474839... and the sent name is Admin | PASS |
Validate User is able to login to the application | FAIL |
=====
Test1
=====
1 test, 1 passed, 0 failed
=====
Output: /Users/yogashivamathivanan/PycharmProjects/Frameworks/RobotFrameworkDemo/Tests/output.xml
Log:   /Users/yogashivamathivanan/PycharmProjects/Frameworks/RobotFrameworkDemo/Tests/log.html
Report: /Users/yogashivamathivanan/PycharmProjects/Frameworks/RobotFrameworkDemo/Tests/report.html
(venv) yogashivamathivanan@Yogashivas-MacBook-Pro Tests %
```

Figure 9.21: Execute test using command Line in robot framework

You can also set the edit configuration in PyCharm, as shown in the following *Figure 9.22*:

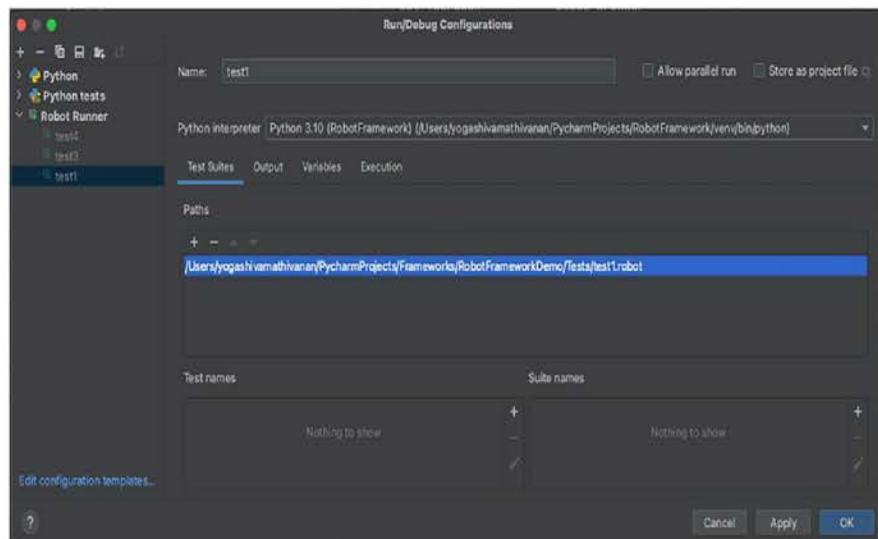


Figure 9.22: Execute test PyCharm in robot framework

Test teardown/Test suite teardown

After all the test cases have been executed, the Robot framework executed the test teardown or test suite teardown as per the test requirement. The teardown is responsible for cleaning up and disconnecting any resources that were used during the test execution. This can be closing the browser, deleting temp files, disconnecting integrations, and so on.

CODE 1 – test1.robot

```
1. *** settings ***
2. Library           SeleniumLibrary
3. Library           ${EXECDIR}/Libraries/Python1.py
                      name=${name}
4. Resource          ${EXECDIR}/resources/resource1.resource
5. Resource          ${EXECDIR}/resources/coreResource.resource
6. Test Setup        Open Application
7. Test Teardown    Close Browser
8.
9. *** variables ***
10. ${url}           https://magento.softwaretestingboard.com/
                      customer/account/login
11. ${username}      admin12@gmail.com
12. ${password}      admin12!@
13. ${name}          Admin
14.
15. *** Test Cases ***
16. Validate User is able to login to the application
17. Given User is able to navigate to the application
18. And Enter the valid login credentials and submit
19. Then Validate user is successfully logged into the application
20. And Get Todays date and time
```

CODE 2– coreResource.resource

```
1. *** Keywords ***
2. Open Application
3. Open Browser       browser=chrome
```

CODE 3 – resource1.resource

```
1. *** Keywords ***
2. User is able to navigate to the application
3. Goto               ${url}
4.
```

5. Enter the valid login credentials and submit

6. Input Text email \${username}

7. Input Password pass \${password}

8. Click Element xpath://*[@id="send2"]

9.

10. Validate user is successfully logged into the application

11. Sleep 5s

12. \${pageTitle} GetTitle

13. Log \${pageTitle}

14. Should be Equal \${pageTitle} My Account Magento
Commerce - website to
practice selenium |
demo website for auto-
mation testing | sele-
nium practice sites

15.

16. Get Todays date and time

17. \${dateTime} Python1.todaysTime

18. Log to Console \${dateTime}

CODE 4 – Python1.py

```

1. import datetime as dt
2.
3.
4. class Python1:
5.
6.     def __init__(self, name):
7.         self.name = name
8.
9.     def todaysTime(self):
10.        dated = str(dt.datetime.today())
11.        return "today's date and time is " + dated + " and the
12.        sent name is " + self.name
13.
14. if __name__ == "__main__":
15.     print(Python1("shiva").todaysTime())

```

Logs and reports in Robot framework

The logs and reports are generated automatically in the robot framework. When executed using the command `robot test1.robot`, the logs and reports are placed in the root directory, to place it in a specific folder, navigate to the root directory where the test results folder is and run the test suite using command `robot --outputdir Results Tests/test1.robot`.

Let us know more about Logs and Reports:

- **Logs:** Logs contain detailed information about each keyword with logged information during test execution. The log file contains the test execution summary, as shown in *Figure 9.23*. By default, the log file name is `log.html`.

Test1 Log		REPORT																																																																																
Generated 2023/02/24 12:57:11 UTC-08:00 11 minutes 39 seconds ago																																																																																		
Test Statistics																																																																																		
<table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th>Total Statistics</th><th>Total</th><th>Pass</th><th>Fail</th><th>Skip</th><th>Elapsed</th><th>Pass / Fail / Skip</th></tr> </thead> <tbody> <tr> <td>All Tests</td><td>1</td><td>1</td><td>0</td><td>0</td><td>00:00:10</td><td>PSS</td></tr> </tbody> </table>			Total Statistics	Total	Pass	Fail	Skip	Elapsed	Pass / Fail / Skip	All Tests	1	1	0	0	00:00:10	PSS																																																																		
Total Statistics	Total	Pass	Fail	Skip	Elapsed	Pass / Fail / Skip																																																																												
All Tests	1	1	0	0	00:00:10	PSS																																																																												
<table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th>Statistics by Tag</th><th>Total</th><th>Pass</th><th>Fail</th><th>Skip</th><th>Elapsed</th><th>Pass / Fail / Skip</th></tr> </thead> <tbody> <tr> <td>No Tags</td><td></td><td></td><td></td><td></td><td></td><td></td></tr> </tbody> </table>			Statistics by Tag	Total	Pass	Fail	Skip	Elapsed	Pass / Fail / Skip	No Tags																																																																								
Statistics by Tag	Total	Pass	Fail	Skip	Elapsed	Pass / Fail / Skip																																																																												
No Tags																																																																																		
<table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th>Statistics by Suite</th><th>Total</th><th>Pass</th><th>Fail</th><th>Skip</th><th>Elapsed</th><th>Pass / Fail / Skip</th></tr> </thead> <tbody> <tr> <td>Test1</td><td>1</td><td>1</td><td>0</td><td>0</td><td>00:00:11</td><td>PSS</td></tr> </tbody> </table>			Statistics by Suite	Total	Pass	Fail	Skip	Elapsed	Pass / Fail / Skip	Test1	1	1	0	0	00:00:11	PSS																																																																		
Statistics by Suite	Total	Pass	Fail	Skip	Elapsed	Pass / Fail / Skip																																																																												
Test1	1	1	0	0	00:00:11	PSS																																																																												
Test Execution Log																																																																																		
<table border="1" style="width: 100%; border-collapse: collapse;"> <tr> <td>- TEST Test1</td><td style="text-align: right;">00:00:10.539</td></tr> <tr> <td> Full Name:</td><td>Test1</td></tr> <tr> <td> Source:</td><td>/Users/yogashivamathivanan/PycharmProjects/framework/RobotFramework/DemoTests/test1.robot</td></tr> <tr> <td> Start / End / Elapsed:</td><td>2023/02/24 12:57:00.801 / 2023/02/24 12:57:11.399 / 00:00:10.598</td></tr> <tr> <td> Status:</td><td>1 test total, 1 passed, 0 failed, 0 skipped</td></tr> <tr> <td>- TEST Validate User is able to login to the application</td><td style="text-align: right;">00:00:10.479</td></tr> <tr> <td> Full Name:</td><td>Test1.Validate User is able to login to the application</td></tr> <tr> <td> Start / End / Elapsed:</td><td>2023/02/24 12:57:00.928 / 2023/02/24 12:57:02.974 / 00:00:02.045</td></tr> <tr> <td> Status:</td><td>PASS</td></tr> <tr> <td> - KEYWORD selenium.Open Application</td><td style="text-align: right;">00:00:02.045</td></tr> <tr> <td> Start / End / Elapsed:</td><td>2023/02/24 12:57:02.850 / 2023/02/24 12:57:02.873 / 00:00:00.243</td></tr> <tr> <td> Documentation:</td><td>Opens a new browser instance to the optional url.</td></tr> <tr> <td> Start / End / Elapsed:</td><td>2023/02/24 12:57:02.850 / 2023/02/24 12:57:02.873 / 00:00:00.243</td></tr> <tr> <td> INFO:</td><td>Opening browser 'chrome' to have url 'None'.</td></tr> <tr> <td> - KEYWORD selenium.Given User is able to navigate to the application</td><td style="text-align: right;">00:00:01.768</td></tr> <tr> <td> Start / End / Elapsed:</td><td>2023/02/24 12:57:02.974 / 2023/02/24 12:57:04.782 / 00:00:01.798</td></tr> <tr> <td> Documentation:</td><td>Navigates the current browser window to the provided url.</td></tr> <tr> <td> Start / End / Elapsed:</td><td>2023/02/24 12:57:02.974 / 2023/02/24 12:57:04.782 / 00:00:01.798</td></tr> <tr> <td> INFO:</td><td>Opening url 'https://magento.softwaretestingboard.com/customer/account/login'</td></tr> <tr> <td> - KEYWORD selenium.And Enter the valid login credentials and submit</td><td style="text-align: right;">00:00:01.524</td></tr> <tr> <td> Start / End / Elapsed:</td><td>2023/02/24 12:57:04.742 / 2023/02/24 12:57:06.266 / 00:00:01.524</td></tr> <tr> <td> Documentation:</td><td>Fills the given text into the text field identified by locator.</td></tr> <tr> <td> Start / End / Elapsed:</td><td>2023/02/24 12:57:04.742 / 2023/02/24 12:57:06.266 / 00:00:01.524</td></tr> <tr> <td> INFO:</td><td>Typing text 'adminaz12@gmail.com' into text field 'email'</td></tr> <tr> <td> - KEYWORD selenium.Input Password pass {Password}</td><td style="text-align: right;">00:00:00.127</td></tr> <tr> <td> - KEYWORD selenium.Click Element {xpath}[@id='send2']</td><td style="text-align: right;">00:00:00.248</td></tr> <tr> <td> - KEYWORD selenium.Then Validate user is successfully logged into the application</td><td style="text-align: right;">00:00:00.309</td></tr> <tr> <td> Start / End / Elapsed:</td><td>2023/02/24 12:57:06.266 / 2023/02/24 12:57:11.275 / 00:00:00.009</td></tr> <tr> <td> Documentation:</td><td>Fails if the given objects are unequal.</td></tr> <tr> <td> Start / End / Elapsed:</td><td>2023/02/24 12:57:11.274 / 2023/02/24 12:57:11.275 / 00:00:00.001</td></tr> <tr> <td> INFO:</td><td>And Get Todays date and time</td></tr> <tr> <td> - KEYWORD selenium.And Get Todays date and time</td><td style="text-align: right;">00:00:00.301</td></tr> <tr> <td> Start / End / Elapsed:</td><td>2023/02/24 12:57:11.275 / 2023/02/24 12:57:11.276 / 00:00:00.001</td></tr> <tr> <td> Documentation:</td><td>Gets the current date and time.</td></tr> <tr> <td> Start / End / Elapsed:</td><td>2023/02/24 12:57:11.275 / 2023/02/24 12:57:11.276 / 00:00:00.001</td></tr> <tr> <td> INFO:</td><td>\$({dataTime}) = \${dataTime}. Todays Time</td></tr> <tr> <td> - KEYWORD selenium.Log \$({pageTitle})</td><td style="text-align: right;">00:00:00.000</td></tr> <tr> <td> - KEYWORD selenium.Should Be Equal \${pageTitle}, My Account Magento Commerce - website to practice selenium demo website for automation testing selenium practice sites</td><td style="text-align: right;">00:00:00.001</td></tr> <tr> <td> Documentation:</td><td>Fails if the given objects are unequal.</td></tr> <tr> <td> Start / End / Elapsed:</td><td>2023/02/24 12:57:11.274 / 2023/02/24 12:57:11.275 / 00:00:00.001</td></tr> <tr> <td> INFO:</td><td>Close Browser</td></tr> </table>	- TEST Test1	00:00:10.539	Full Name:	Test1	Source:	/Users/yogashivamathivanan/PycharmProjects/framework/RobotFramework/DemoTests/test1.robot	Start / End / Elapsed:	2023/02/24 12:57:00.801 / 2023/02/24 12:57:11.399 / 00:00:10.598	Status:	1 test total, 1 passed, 0 failed, 0 skipped	- TEST Validate User is able to login to the application	00:00:10.479	Full Name:	Test1.Validate User is able to login to the application	Start / End / Elapsed:	2023/02/24 12:57:00.928 / 2023/02/24 12:57:02.974 / 00:00:02.045	Status:	PASS	- KEYWORD selenium.Open Application	00:00:02.045	Start / End / Elapsed:	2023/02/24 12:57:02.850 / 2023/02/24 12:57:02.873 / 00:00:00.243	Documentation:	Opens a new browser instance to the optional url.	Start / End / Elapsed:	2023/02/24 12:57:02.850 / 2023/02/24 12:57:02.873 / 00:00:00.243	INFO:	Opening browser 'chrome' to have url 'None'.	- KEYWORD selenium.Given User is able to navigate to the application	00:00:01.768	Start / End / Elapsed:	2023/02/24 12:57:02.974 / 2023/02/24 12:57:04.782 / 00:00:01.798	Documentation:	Navigates the current browser window to the provided url.	Start / End / Elapsed:	2023/02/24 12:57:02.974 / 2023/02/24 12:57:04.782 / 00:00:01.798	INFO:	Opening url ' https://magento.softwaretestingboard.com/customer/account/login '	- KEYWORD selenium.And Enter the valid login credentials and submit	00:00:01.524	Start / End / Elapsed:	2023/02/24 12:57:04.742 / 2023/02/24 12:57:06.266 / 00:00:01.524	Documentation:	Fills the given text into the text field identified by locator.	Start / End / Elapsed:	2023/02/24 12:57:04.742 / 2023/02/24 12:57:06.266 / 00:00:01.524	INFO:	Typing text 'adminaz12@gmail.com' into text field 'email'	- KEYWORD selenium.Input Password pass {Password}	00:00:00.127	- KEYWORD selenium.Click Element {xpath}[@id='send2']	00:00:00.248	- KEYWORD selenium.Then Validate user is successfully logged into the application	00:00:00.309	Start / End / Elapsed:	2023/02/24 12:57:06.266 / 2023/02/24 12:57:11.275 / 00:00:00.009	Documentation:	Fails if the given objects are unequal.	Start / End / Elapsed:	2023/02/24 12:57:11.274 / 2023/02/24 12:57:11.275 / 00:00:00.001	INFO:	And Get Todays date and time	- KEYWORD selenium.And Get Todays date and time	00:00:00.301	Start / End / Elapsed:	2023/02/24 12:57:11.275 / 2023/02/24 12:57:11.276 / 00:00:00.001	Documentation:	Gets the current date and time.	Start / End / Elapsed:	2023/02/24 12:57:11.275 / 2023/02/24 12:57:11.276 / 00:00:00.001	INFO:	\$({dataTime}) = \${dataTime}. Todays Time	- KEYWORD selenium.Log \$({pageTitle})	00:00:00.000	- KEYWORD selenium.Should Be Equal \${pageTitle}, My Account Magento Commerce - website to practice selenium demo website for automation testing selenium practice sites	00:00:00.001	Documentation:	Fails if the given objects are unequal.	Start / End / Elapsed:	2023/02/24 12:57:11.274 / 2023/02/24 12:57:11.275 / 00:00:00.001	INFO:	Close Browser
- TEST Test1	00:00:10.539																																																																																	
Full Name:	Test1																																																																																	
Source:	/Users/yogashivamathivanan/PycharmProjects/framework/RobotFramework/DemoTests/test1.robot																																																																																	
Start / End / Elapsed:	2023/02/24 12:57:00.801 / 2023/02/24 12:57:11.399 / 00:00:10.598																																																																																	
Status:	1 test total, 1 passed, 0 failed, 0 skipped																																																																																	
- TEST Validate User is able to login to the application	00:00:10.479																																																																																	
Full Name:	Test1.Validate User is able to login to the application																																																																																	
Start / End / Elapsed:	2023/02/24 12:57:00.928 / 2023/02/24 12:57:02.974 / 00:00:02.045																																																																																	
Status:	PASS																																																																																	
- KEYWORD selenium.Open Application	00:00:02.045																																																																																	
Start / End / Elapsed:	2023/02/24 12:57:02.850 / 2023/02/24 12:57:02.873 / 00:00:00.243																																																																																	
Documentation:	Opens a new browser instance to the optional url.																																																																																	
Start / End / Elapsed:	2023/02/24 12:57:02.850 / 2023/02/24 12:57:02.873 / 00:00:00.243																																																																																	
INFO:	Opening browser 'chrome' to have url 'None'.																																																																																	
- KEYWORD selenium.Given User is able to navigate to the application	00:00:01.768																																																																																	
Start / End / Elapsed:	2023/02/24 12:57:02.974 / 2023/02/24 12:57:04.782 / 00:00:01.798																																																																																	
Documentation:	Navigates the current browser window to the provided url.																																																																																	
Start / End / Elapsed:	2023/02/24 12:57:02.974 / 2023/02/24 12:57:04.782 / 00:00:01.798																																																																																	
INFO:	Opening url ' https://magento.softwaretestingboard.com/customer/account/login '																																																																																	
- KEYWORD selenium.And Enter the valid login credentials and submit	00:00:01.524																																																																																	
Start / End / Elapsed:	2023/02/24 12:57:04.742 / 2023/02/24 12:57:06.266 / 00:00:01.524																																																																																	
Documentation:	Fills the given text into the text field identified by locator.																																																																																	
Start / End / Elapsed:	2023/02/24 12:57:04.742 / 2023/02/24 12:57:06.266 / 00:00:01.524																																																																																	
INFO:	Typing text 'adminaz12@gmail.com' into text field 'email'																																																																																	
- KEYWORD selenium.Input Password pass {Password}	00:00:00.127																																																																																	
- KEYWORD selenium.Click Element {xpath}[@id='send2']	00:00:00.248																																																																																	
- KEYWORD selenium.Then Validate user is successfully logged into the application	00:00:00.309																																																																																	
Start / End / Elapsed:	2023/02/24 12:57:06.266 / 2023/02/24 12:57:11.275 / 00:00:00.009																																																																																	
Documentation:	Fails if the given objects are unequal.																																																																																	
Start / End / Elapsed:	2023/02/24 12:57:11.274 / 2023/02/24 12:57:11.275 / 00:00:00.001																																																																																	
INFO:	And Get Todays date and time																																																																																	
- KEYWORD selenium.And Get Todays date and time	00:00:00.301																																																																																	
Start / End / Elapsed:	2023/02/24 12:57:11.275 / 2023/02/24 12:57:11.276 / 00:00:00.001																																																																																	
Documentation:	Gets the current date and time.																																																																																	
Start / End / Elapsed:	2023/02/24 12:57:11.275 / 2023/02/24 12:57:11.276 / 00:00:00.001																																																																																	
INFO:	\$({dataTime}) = \${dataTime}. Todays Time																																																																																	
- KEYWORD selenium.Log \$({pageTitle})	00:00:00.000																																																																																	
- KEYWORD selenium.Should Be Equal \${pageTitle}, My Account Magento Commerce - website to practice selenium demo website for automation testing selenium practice sites	00:00:00.001																																																																																	
Documentation:	Fails if the given objects are unequal.																																																																																	
Start / End / Elapsed:	2023/02/24 12:57:11.274 / 2023/02/24 12:57:11.275 / 00:00:00.001																																																																																	
INFO:	Close Browser																																																																																	

 | |



Figure 9.23: log.html in robot framework

- The test execution **report** file is generated while execution, and provides a summary of the test suite execution. By default, the report file is named “**report.html**”. The report file contains the Test case name, status, execution time, links to the log file, and a summary of the test suite execution. The report file is shown in *Figure 9.24*:

Test1 Report		LOG														
Generated 2023/02/24 12:57:11 UTC-08:00 36 minutes 20 seconds ago																
Summary Information																
Status: All tests passed Start Time: 2023/02/24 12:57:00.801 End Time: 2023/02/24 12:57:11.399 Elapsed Time: 00:00:10.598 Log File: log.html																
Test Statistics																
<table border="1"><thead><tr><th>Total Statistics</th><th>Total</th><th>Pass</th><th>Fail</th><th>Skip</th><th>Elapsed</th><th>Pass / Fail / Skip</th></tr></thead><tbody><tr><td>All Tests</td><td>1</td><td>1</td><td>0</td><td>0</td><td>00:00:10</td><td>GREEN</td></tr></tbody></table>			Total Statistics	Total	Pass	Fail	Skip	Elapsed	Pass / Fail / Skip	All Tests	1	1	0	0	00:00:10	GREEN
Total Statistics	Total	Pass	Fail	Skip	Elapsed	Pass / Fail / Skip										
All Tests	1	1	0	0	00:00:10	GREEN										
<table border="1"><thead><tr><th>Statistics by Tag</th><th>Total</th><th>Pass</th><th>Fail</th><th>Skip</th><th>Elapsed</th><th>Pass / Fail / Skip</th></tr></thead><tbody><tr><td>No Tags</td><td></td><td></td><td></td><td></td><td></td><td></td></tr></tbody></table>			Statistics by Tag	Total	Pass	Fail	Skip	Elapsed	Pass / Fail / Skip	No Tags						
Statistics by Tag	Total	Pass	Fail	Skip	Elapsed	Pass / Fail / Skip										
No Tags																
<table border="1"><thead><tr><th>Statistics by Suite</th><th>Total</th><th>Pass</th><th>Fail</th><th>Skip</th><th>Elapsed</th><th>Pass / Fail / Skip</th></tr></thead><tbody><tr><td>Test1</td><td>1</td><td>1</td><td>0</td><td>0</td><td>00:00:11</td><td>GREEN</td></tr></tbody></table>			Statistics by Suite	Total	Pass	Fail	Skip	Elapsed	Pass / Fail / Skip	Test1	1	1	0	0	00:00:11	GREEN
Statistics by Suite	Total	Pass	Fail	Skip	Elapsed	Pass / Fail / Skip										
Test1	1	1	0	0	00:00:11	GREEN										
Test Details																
<table border="1"><thead><tr><th>All</th><th>Tags</th><th>Suites</th><th>Search</th></tr></thead><tbody><tr><td>Status:</td><td colspan="3">1 test total, 1 passed, 0 failed, 0 skipped</td></tr><tr><td>Total Time:</td><td colspan="3">00:00:10.470</td></tr></tbody></table>			All	Tags	Suites	Search	Status:	1 test total, 1 passed, 0 failed, 0 skipped			Total Time:	00:00:10.470				
All	Tags	Suites	Search													
Status:	1 test total, 1 passed, 0 failed, 0 skipped															
Total Time:	00:00:10.470															
<table border="1"><thead><tr><th>Name</th><th>Documentation</th><th>Tags</th><th>Status</th><th>Message</th><th>Elapsed</th><th>Start / End</th></tr></thead><tbody><tr><td>test: Validate User is able to login to the application</td><td></td><td></td><td>PASS</td><td></td><td>00:00:10.470</td><td>2023/02/24 12:57:00.928 2023/02/24 12:57:11.398</td></tr></tbody></table>			Name	Documentation	Tags	Status	Message	Elapsed	Start / End	test: Validate User is able to login to the application			PASS		00:00:10.470	2023/02/24 12:57:00.928 2023/02/24 12:57:11.398
Name	Documentation	Tags	Status	Message	Elapsed	Start / End										
test: Validate User is able to login to the application			PASS		00:00:10.470	2023/02/24 12:57:00.928 2023/02/24 12:57:11.398										

Figure 9.24: reports.html in robot framework

Along with the report and log file, there is also an **output.xml** file generated, that provides the test execution result in XML format, with all the execution details. This is used to merge the test results, especially by build tools such as Jenkins, buildkite, and so on, as well as debugging, troubleshooting and many more usages.

Advantages of Robot framework

The advantages of Robot Framework are as follows:

- Easy to implement:** The robot framework is easy to implement even with little to no programming experience.
- Keyword-driven testing:** Robot framework allows writing the test cases using high-level, business-friendly language. This helps in including non-technical stakeholders in the testing process.

- **Test libraries:** There are inbuild and custom test libraries for all the testing automation requirements. This reduces the amount of code and makes implementation faster and easier to create complex test suites.

- **Multiple platform support:** The robot framework can be used in any operating system that supports Python.
- **Integration with external tools:** The robot framework can easily be integrated with other tools and frameworks, which makes the framework highly versatile. Moreover, the robot framework can be integrated with CI/CD pipelines for automated testing for the build and release process.

Unittest

The unittest testing framework is Python's inbuilt module for unit testing. It provides tools to write and run the test. This does not require any installation as it is part of Python's standard library. To use the methods that are present in the `UnitTest` framework, we are required to import the module and extend the `TestCase` class in the `Unittest` module.

The `unittest` framework works by providing a set of test classes and test methods that we can leverage to write the test code. A test class is a Python class that contains test methods, and a test method is a Python method that checks specific units of the complete code.

Here is an example of the basic structure of the unit test. In the following **CODE 1**, the class `UnitDemo` inherits from '`unittest.TestCase`' class, which provides the basic structure for writing tests. The `setUp()` method is called before test and executed before each test. Similarly, the `tearDown()` method is called after test, and executed after each test. When the `test_function()` method is executed, the `setUp()` and `tearDown()` methods are also executed before and after the test. Similarly, there are `setUpClass()` and `tearDownClass()` methods to be executed before and after the class, as well as the `setUpModule()` and `tearDownModule()` methods to be executed before and after the module."

In another example in **CODE 2**, multiple tests can be executed using `unittest.main()`. A test case is used to define a set of tests that you want to run. In the example, we define a test case called `PageTitleTest` which contains three tests, one `setUp()` and one `tearDown()` method. The `setUp()` method initializes the Chrome browser, the `tearDown()` method closes the browser using `driver.quit()`, and the tests `test_verifyAmazonHomeTitle`, `test_verifyMacysHomeTitle` and `test_`

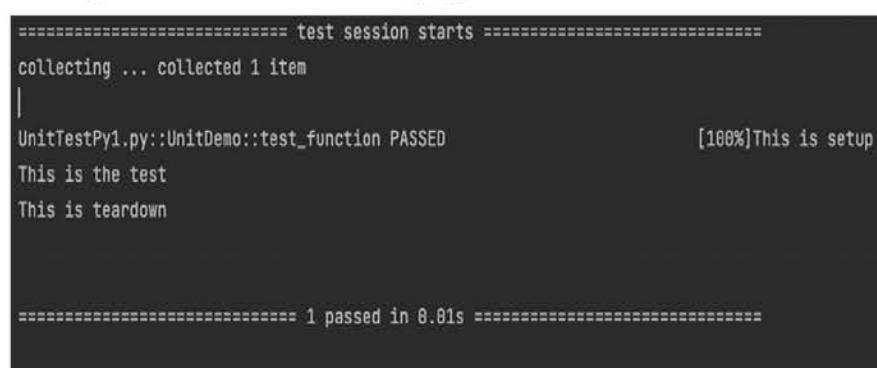
verifyEbayHomeTitle verify the title of the home page using assert. Additionally, a test can be skipped with a condition using annotation `@unittest.skip` or `@unittest.skipIf(condition, message)`.

CODE 1

```
1. import unittest
2.
3.
4. class UnitDemo(unittest.TestCase):
5.
6.     def setUp(self) -> None:
7.         print("This is setup")
8.
9.     def test_function(self):
10.        print("This is the test")
11.
12.    def tearDown(self) -> None:
13.        print("This is teardown")
14.
15.
16. if __name__ == "__main__":
17.     UnitDemo.test_function()
```

OUTPUT 1

The output can be seen in the following *Figure 9.25*:



```
===== test session starts =====
collecting ... collected 1 item
|
UnitTestPy1.py::UnitDemo::test_function PASSED [100%]This is setup
This is the test
This is teardown

===== 1 passed in 0.01s =====
```

A terminal window showing the execution of a Python unit test. The command `python -m unittest UnitTestPy1.py` is run, and the output shows the test suite collecting one item, running the `test_function` test, and printing the setup, test, and teardown messages. The test passes with a duration of 0.01 seconds.

```
Process finished with exit code 0
```

Figure 9.25: Output

CODE 2

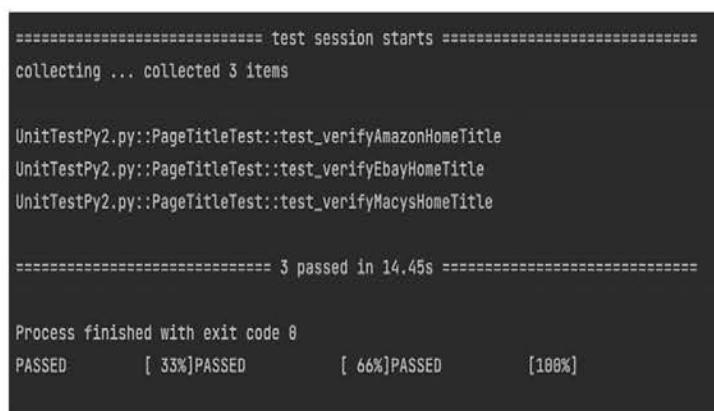
```
1. import unittest
2. from selenium import webdriver
3. from selenium.webdriver.chrome.service import Service
4. from webdriver_manager.chrome import ChromeDriverManager
5.
6.
7. class PageTitleTest(unittest.TestCase):
8.
9.     def setUp(self) -> None:
10.         self.driver = webdriver.Chrome(service=Service(
11.             ChromeDriverManager("").install()))
12.
13.     def test_verifyAmazonHomeTitle(self):
14.         self.driver.get("http://www.amazon.com")
15.         assert "Amazon.com" in self.driver.title
16.
17.     def test_verifyMacysHomeTitle(self):
18.         self.driver.get("http://www.macys.com")
19.         assert "Macy's" in self.driver.title
20.
21.     # @unittest.skip("Reason")
22.     @unittest.skipIf(1!=1, "Number is equal")
23.     def test_verifyEbayHomeTitle(self):
24.         self.driver.get("http://www.ebay.com")
```

```
25.         assert "eBay" in self.driver.title
26.
27.     def tearDown(self) -> None:
28.         self.driver.quit()
29.
30.
```

```
31. if __name__ == '__main__':
32.     unittest.main()
33.
```

OUTPUT 2

The output can be seen in the following *Figure 9.26*:



```
===== test session starts =====
collecting ... collected 3 items

UnitTestPy2.py::PageTitleTest::test_verifyAmazonHomeTitle
UnitTestPy2.py::PageTitleTest::test_verifyEbayHomeTitle
UnitTestPy2.py::PageTitleTest::test_verifyMacysHomeTitle

===== 3 passed in 14.45s =====

Process finished with exit code 0
PASSED      [ 33%]PASSED          [ 66%]PASSED        [100%]
```

Figure 9.26: Output

Behave

Behave is a **Behavior Driven Development (BDD)** framework for Python that enables writing and executing high-level scenarios and feature files in a natural language format. It is built on top of Gherkin language, which is written using “Given”, “When”, “And” and “Then” keywords, which are easy to understand by both technical and non-technical stakeholders.

Behavior Driven Development Keywords

As mentioned earlier, **Behavior Driven Development (BDD)** framework creates a culture where testers, developers, product owner, and other project stakeholders can collaborate towards the software development. Following is an example of feature file with BDD structure.

In the following code, the feature file defines two scenarios: one for valid credentials and one for invalid credentials. The steps in each scenario are written in a natural language format that describes the actions to be performed. Given, When, and Then are keywords used to define the steps in these scenarios. Here is a brief explanation of what each keyword means:

- **Given:** The Given step is used to describe the preconditions or the initial state of the system. It sets up the context in which the scenario takes place. In the following code, the Given step is used to navigate to the login page.

- **When:** The When step is used to describe the action or event that the user performs on the system. It represents the trigger that causes the system to behave in a certain way. In the following code, the When step is used to input the username and password. Then click the login button.
- **Then:** The Then step is used to describe the expected outcome or result of the action or event described in the When step. It represents the expected behavior of the system. In this example, the Then step is used to check whether the user is redirected to the dashboard page or whether an error message is displayed.
- **And:** The And step is used to concatenate multiple Given, When or Then steps, that belong to the same scenario and create more complex scenarios that involve multiple steps, making it easier to write more comprehensive tests. In the following code, the And step is used to add multiple When step.

CODE – login.feature

1. Feature: Login functionality
2. As a user,
3. I want to be able to log in to the website,
4. So that I can access my account.
- 5.
6. Scenario: Valid credentials
7. Given I am on the login page
8. When I enter valid username and password
9. And I click the login button
10. Then I should see the dashboard page
- 11.
12. Scenario: Invalid credentials
13. Given I am on the login page
14. When I enter invalid username and password
15. And I click the login button

Behave environment and project setup

Behave requires Python and we can install Behave using the pip command `pip install behave`. Once the installation is complete, verify using `behave -version` command. Install Selenium and create a directory “features” in the project directory. This directory will contain all the feature files of our project, and one of the feature files is given in the preceding code. The feature files will have the extension “.feature” and when the file is created with “.feature” extension, we will be prompted

to install Gherkin plugin, which provides support for editing and running Gherkin feature files. Create new directory “steps” within feature or project directory, and this directory will contain the step definitions that contain the feature implementation that correspond to the steps in the feature file. Step definition will have files with “.py” extension.

Behave step definitions and implementation

To create the step definition corresponding to the features, we can use command `behave -dry-run` or just type `behave` in the project directory, or “`behave [path to feature file]`” like “`behave features/login.feature`”. This will output the unimplemented feature files step definitions to the console in a specific format as in *Figure 9.27*. We can copy these step definitions into the step definition file to create the necessary step definition function.

```
(venv) yogashivamathivanangregashivas-MacBook-Pro BehaveFrameworkDemo % behave features/login.feature
Feature: Login functionality # features/login.feature:1
  As a user,
  I want to be able to log in to the website,
  so that I can access my account.

Scenario: Valid credentials # features/login.feature:4
  Given I am on the login page # None
  When I enter valid username and password # None
  And I click the login button # None
  Then I should see the dashboard page # None

Scenario: Invalid credentials # features/login.feature:12
  Given I am on the login page # None
  When I enter invalid username and password # None
  And I click the login button # None
  Then I should see an error message # None

Failing scenarios:
  features/login.feature:6  Valid credentials
  features/login.feature:12  Invalid credentials

0 features passed, 1 failed, 0 skipped
0 scenarios passed, 2 failed, 0 skipped
0 steps passed, 0 failed, 0 skipped, 8 undefined
Took 0m0.000s

You can implement step definitions for undefined steps with these snippets:

@given(u'I am on the login page')
```

```

def step_implementation(context):
    raise NotImplementedError("STEP: Given I am on the login page")

@when(u'I enter valid username and password')
def step_implementation(context):
    raise NotImplementedError("STEP: When I enter valid username and password")

@then(u'I click the login button')
def step_implementation(context):
    raise NotImplementedError("STEP: Then I click the login button")

@then(u'I should see the dashboard page')
def step_implementation(context):
    raise NotImplementedError("STEP: Then I should see the dashboard page")

```

Figure 9.27: behave dry run for step definition

Here is the step definition file that contains step definition function corresponding to the steps in feature file and implements the valid and invalid login scenarios.

CODE – appLoginSteps.py

```

1. from behave import *
2. from selenium import webdriver
3. from selenium.webdriver.chrome.service import Service
4. from selenium.webdriver.common.by import By
5. from webdriver_manager.chrome import ChromeDriverManager
6.
7.
8. @given(u'I am on the login page')
9. def launch_browser(context):
10.     context.driver = webdriver.Chrome(service=Service(ChromeDriverManager("").install()))
11.     context.driver.get("https://magento.softwaretestingboard.com/
customer/account/login")
12.
13.
14. @when(u'I enter valid username and password')
15. def valid_username_password(context):
16.     context.driver.find_element(By.CSS_SELECTOR, "#email").send_
keys("admin12@gmail.com")
17.     context.driver.find_element(By.ID, "pass").send_keys("ad-
min12!@")

```

```

18.
19. @when(u'I click the login button')
20. def click_login(context):
21.     context.driver.find_element(By.NAME, "send").click()
22.
23. @then(u'I should see the dashboard page')
24. def verifyHomePage(context):
25.     status = context.driver.find_element(By.XPATH, "//*[@data-ui-
        id='page-title-wrapper']").is_displayed()
26.     assert status is True
27.

```

```

28. @when(u'I enter invalid username and password')
29. def invalid_username_password(context):
30.     context.driver.find_element(By.CSS_SELECTOR, "#email").send_
        keys("admin12invalid@gmail.com")
31.     context.driver.find_element(By.ID, "pass").send_
        keys("admin12!@invalid")
32.
33. @then(u'I should see an error message')
34. def verify_unsuccessful_login(context):
35.     status = context.driver.find_element(By.NAME, "send").is_
        displayed()
36.     assert status is True
37.
38. @then(u'I close the browser')
39. def step_impl(context):
40.     context.driver.quit()

```

Behave Test execution

To execute, type command **behave** for entire feature file execution in the directory or “behave [path to feature file]” for specific feature file execution. The output of the execution is shown in *Figure 9.28*:

```
(venv) yogashivamathanan@Yogashivas-MacBook-Pro ~ BehaveFrameworkDemo % behave
Feature: Login functionality
  As a user,
  I want to be able to log in to the website,
  So that I can access my account.
  Scenario: Valid credentials
```

```

Scenario: Valid credentials
  Given I am on the login page          # features/steps/appLoginSteps.py:18  5.448s
  When I enter valid username and password # features/steps/appLoginSteps.py:16  0.240s
  And I click the login button         # features/steps/appLoginSteps.py:22  1.734s
  Then I should see the dashboard page # features/steps/appLoginSteps.py:27  0.046s
  And I close the browser             # features/steps/appLoginSteps.py:45  0.175s

Scenario: Invalid credentials
  Given I am on the login page          # features/steps/appLoginSteps.py:18  1.010s
  When I enter invalid username and password # features/steps/appLoginSteps.py:18  0.268s
  And I click the login button         # features/steps/appLoginSteps.py:22  1.199s
  Then I should see an error message   # features/steps/appLoginSteps.py:19  0.055s
  And I close the browser             # features/steps/appLoginSteps.py:45  0.118s

1 feature passed, 0 failed, 0 skipped
2 scenarios passed, 0 failed, 0 skipped
10 steps passed, 0 failed, 0 skipped, 0 undefined
Took 8m9.686s
(venv) yogashivamathivanan@Yogashivas-MacBook-Pro BehaveFrameworkDemo %

```

Figure 9.28: behave test execution output

Behave Data Parameters, Data Parameterization and Background

We can pass the data directly from the scenario step in the feature file using data parameterization. For example, the step “**When I enter valid “admin12@gmail.com” and “admin12!@”**” in **CODE1**, it enters a valid username and password. The step definition “`@when(u'I enter valid "{username}" and "{password}"')`” defines the step with two data parameters, username and password, which correspond to the values “admin12@gmail.com” and “admin12!@” respectively. The function “`valid_username_password_parameter`” is the step definition function for this step. The function takes three parameters, context, username, and password, which are automatically passed by Behave when the step is executed.

Similarly, in the following **CODE2**, we can use Scenario Outline to pass multiple values to execute the test multiple times using single scenario. When the following CODE2 will be executed, the test will run three times using the three username and password values passed in the “Examples”.

We can use the “**Background**” keyword to define the steps that are common or precondition for each scenario in a feature file and are executed before every scenario in the feature file.

CODE1 – login.feature & – appLoginSteps.py

1. When I enter valid “admin12@gmail.com” and “admin12!@”
- 2.
-

```

3.
4. @when(u'I enter valid "{username}" and "{password}"')
5. def valid_username_password_parameter(context, username, password):
6.     context.driver.find_element(By.CSS_SELECTOR, "#email").send_
    keys("admin12@gmail.com")
    context.driver.find_element(By.ID, "pass").send_keys("admin12!@")

```

CODE2 – login.feature

1. Scenario Outline: Multiple credentials
2. Given I am on the login page
3. When I enter valid "<username>" and "<password>"
4. And I click the login button
- 5.

6. Examples:
7. |username|password|
8. |admin12@gmail.com|admin12!@|
9. |admin12invalid@gmail.com|admin12!@invalid|
10. |admin12invalid1@gmail.com|admin12!@invalid1|

Difference Between Pytest, Robot framework, Behave and Unittest

pytest, Robot Framework, behave and unittest are all popular testing frameworks used for software testing. The difference is in their syntax, features, and level of complexity. Choosing the right framework depends on your testing needs, project requirements, and personal preferences. *Table 9.2* features the key differences between these frameworks:

Framework	Language	Syntax	Test Discovery	Fixtures	Assertions	Key Features
pytest	Python	Function decorators	Automatic	Yes	Advanced	Parameterization, fixture management, test filtering
Robot Framework	Python (and other languages)	Tabular syntax with keywords	Automatic	Yes	Basic	Versatility, keyword-driven testing, test libraries

unittest	Python	Test classes and methods	Manual or Automatic	Yes	Basic	Simplicity, basic infrastructure for writing tests
Behave	Python	Gherkin	Automatic	Yes	Basic	Behavior-driven development (BDD), readability, team collaboration.

Table 9.2: Differences between pytest, robot framework and unittest

Conclusion

The automation framework is the core of test automation. Automating application features within a framework can achieve stability, flexibility, maintainability, organized structure, efficiency, and so on. The Automation framework is a combination of tools

in sync, to bring out efficient automation for the application. The significance of the automation framework is vast enough that it cannot be ignored when we talk about the automation of applications. Along with discussing the benefits and significance of automation framework, we also discussed the types of automation frameworks which include Linear Scripting Framework, Modular Testing Framework, Data-Driven Testing Framework, Keyword-Driven Testing Framework, Behavior-Driven Development Framework, and Hybrid Testing Framework.

Choosing the right framework depends on the testing requirement, stakeholders of the testing efforts, cost, time flexibility, and so on. Then we had a detailed discussion of the Pytest test framework. Pytest is used for functional, unit, and integration testing automation. Pytest is easy to use, and fixtures help in reducing code duplication. Pytest asserts statements, plugins, Parameterization, integration with other tools, reports, and so on. Then we discussed the Robot framework which is often used for acceptance testing. The robot framework supports a variety of testing approaches, including keyword-driven testing, data-driven testing, and **Behavior-Driven Testing (BDD)**. The strength of the robot framework lies in its support for rich libraries. Then we discussed the Python inbuild test automation framework unittest. The framework plays a major role and is a vast topic of discussion in test automation. Finally, we discussed another majorly used framework that leverages BDD and so we will continue the discussion on the framework in the areas of how to apply the learnings of automation framework from this chapter, to achieve efficiency and other advantages of automation framework.

Key facts

- The automation framework can help manage the complexity of the testing process and efficiently execute all the automated tests. The framework might include, Tool/Library, Test data management, Logging and Reporting, Test case management, and Integration support with other tools.
- The benefits of an automation framework are Scalability and Modularity, Maintenance and Cost Effectiveness, Reusability, Documentation, Ease of Scripting, Understandability, Integration, and Reporting.
- Different types of automation frameworks are Linear Scripting Framework, Modular Testing Framework, Data-Driven Testing Framework, Keyword-Driven Testing Framework, Behavior-Driven Development Framework, and Hybrid Testing Framework. Choosing the right framework depends on the testing requirement, stakeholders of the testing efforts, cost, time flexibility, and so on.

-
- Choose the suitable Automation Framework, based on the factors such as testing needs, cost, community support, and flexibility.
 - Pytest is one of the most popular and widely used open-source Python testing frameworks that is simple and flexible to handle complex testing scenarios.
 - File and function names in pytest should follow the naming convention and should start with 'test'.
 - Markers in pytest are a way of associating metadata with test functions or test classes. Markers are used to categorize tests or to perform actions on tests based on their metadata.
 - Pytest fixtures are functions that define a baseline or a set of preconditions for a test. These functions provide reusable code that can be executed before a test and/or after a test. Essentially, fixtures serve the purpose of setup and teardown functions for a test.
 - In pytest, you can also parameterize tests using fixtures and the `params` keyword. This allows you to define a fixture that generates a set of input values, and then use those values to parameterize one or more test functions.
 - HTML reports can be generated for the test runs in Pytest using the plugin `pytest-html`.
 - The robot framework is well suited for acceptance testing, **Acceptance Test Driven Development (ATDD)**, **Behavior Driven Development (BDD)**, and

Robotic Process Automation (RPA). It is designed to write test cases in a keyword-driven and simple testing format.

- Robot Framework Test Execution Flow starts from Test Setup/ Test Suite Setup, then the test case execution followed by Test Teardown/ Test Suite Teardown.
- Robot framework auto-generates log and report files.
- The unittest testing framework is Python's inbuilt module for unit testing.

Questions

1. What are the significance and advantages of automation framework in test automation?
2. What are the different types of automation frameworks and what are the advantages and disadvantages of each framework?
3. What is Pytest and why is it used for? What are Markers and Fixtures in Pytest? How can we define scope in fixture?

4. How to generate html report of executed tests in pytest?
5. What are a robot framework and its advantages?
6. What is a unittest framework and how the execution happens in the framework?
7. What is behave framework and BDD? What is the significance of keyword "Scenario Outline" and "Background"?

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



CHAPTER 10

Selenium

Automation

Framework – Part 2

Introduction

In the previous chapter, we discussed the significance of the framework. In the rapidly growing software technology, a well-structured framework can reduce complexity, improve efficiency, and accuracy, and speed up the application automation development process. A test automation framework can help organize test cases, manage test data, and generate reports. We will continue the discussion with the frameworks and look at the best practices that help simplify the code implementation within the framework, such as Page Object Model. The **Page Object Model (POM)** is a widely adopted design pattern used in Selenium automation testing, that makes the tests to be more modular, maintainable, and reusable. We will continue to use the pytest framework, which is the most popular and widely used framework for Selenium automation to implement the page object model.

The growing complexity of applications increases the complexity around test automation use cases or scenarios. With a wide range of libraries and packages offered by the powerful and versatile programming language, Python helps with numerous such complex use cases. In this chapter, we will discuss some of the essential Python packages that are commonly used in test automation. To manipulate, transform, compare, and work with data from various sources such as CSV, and SQL database,

Pandas library can be used to read and write data. To generate and compute large data, we can use the NumPy library. In addition to that, Python offers other packages and libraries such as os, pathlib, logging, datetime, random, and so on. The os and pathlib libraries are commonly used to interact with operating systems and help with the file system, directories, and config files, where environment variables are stored. Logging provides a flexible logging system to analyze test output. These libraries sit on top of the test automation framework to help reduce the complexity. Python package random helps generate random test data.

Furthermore, apart from reducing the complexity, the two major parts of good automation are generating reports and the ability to execute the tests in parallel. In the previous chapter, we saw generating pytest-html report; in this chapter, we will also look at allure-pytest which generates test reports in the allure format. We can use the **pytest-xdist** pytest plugin that provides support for parallel test execution.

Structure

In this chapter, we will discuss the following topic:

- Page Object Model (POM)
- Implementing Page Object Model
- Screenshots
- PyAutoGUI
- Configurations
- Logging
- Parallel Test Execution
- Data-Driven Testing with Data source
- NumPy
- Pickle
- DateTime
- Random
- Faker

Objectives

In this chapter, we will continue the discussion in framework development for UI automation using Selenium and Python. We will discuss the implementation of the framework using the **Page Object Model (POM)**. The page object module

helps make the framework more robust. In a given application, we will discuss the best practice in UI automation, using the page object model and how it helps in bringing modularity, maintainability, and reusability to the framework that makes the framework adaptable to new feature changes to the application. In addition to this, we will learn the Python packages and libraries such as Pandas, NumPy, os & pathlib, datetime, and so on, to understand how we can leverage these modules to automate complex test automation use cases.

Page Object Model (POM)

The Page Object Model is a UI automation design pattern that is used to improve the framework in terms of maintainability and reusability. It is an automation design pattern where Python classes are created for each of the pages in the application under test and these Python classes are called Page Objects. The Page Objects contain each element locator and locator actions on the page.

Why Page Object Model?

The application is evolving continuously and there can be continuous addition of new features to the application, which may be re-structuring of a given page, and so on. This will result in the failure of the test cases involving the updated section of the application. To fix the failing automated test, the test engineer has to update all the updated element locators across all the failing tests. In the case of smaller applications, the automation fix update might take up to a day or a week. But for larger applications, the time to fix the updated application can take months, and the team cannot accommodate such a long wait time. This is why the page object model plays a significant role in maintaining automated tests, as it separates the class for each page of the application. Any change in the application will only require an update to happen in the respective page objects. This reduces the risk of errors and it is easier to update the tests. Thus, it improves the maintainability of the automation tests.

Second, the page object model improves the readability of the automated tests by adapting a structure where every page element and element actions are encapsulated in a page object or a Python class. The page class, variables, and functions naming help to identify a specific page, page elements, and element actions. This structure helps the tests to become more modular and thus makes them easier to understand.

Third, the page object model improves the reusability of the automated tests. When the application page is defined using the page object class with the element and element actions, any new application feature that flows through the page does not

require rewriting. The created page object classes can be reused across multiple test cases, reducing duplication, and improving efficiency.

Implementing Page Object Model

Let us implement the POM to understand the benefits. The first step in designing a page object model is to identify the pages and features of the application. The basic prerequisites are to install pytest and Selenium, and as we move forward, we will also make use of other Python packages. To illustrate the page object model, let us consider a sample e-commerce application with pages and various use cases.

Let us start with the project folder structure. The project folder structure is equally important to organize the code and resources, which helps in easy understanding, and maintenance of the code for evolving framework with new features. The project folder structure highly depends on the project requirement and team preference. In *Figure 10.1*, the basic project folder structure is shown:

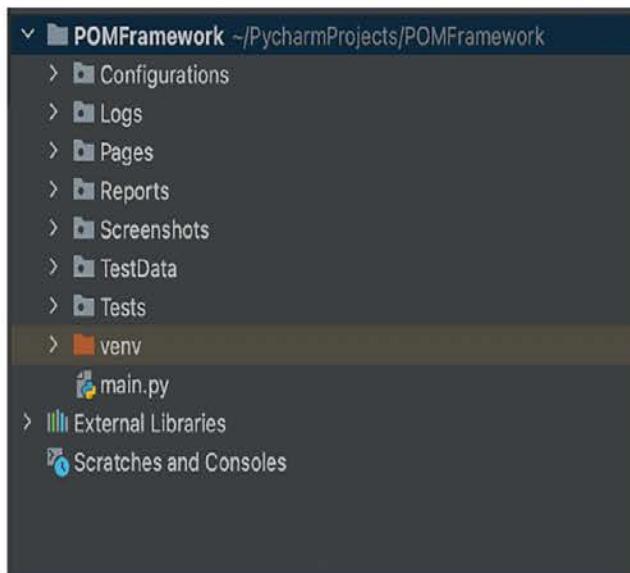


Figure 10.1: Python Framework Folder Structure

The **Tests** folder contains Python files with all the test cases for the project. The test file should follow the naming convention of the framework tests are implemented. In the pytest framework, the test file should start with “**test_**” for pytest to recognize the test file.

The **Pages** folder contains all the page object classes for each of the application pages. Each page object class contains all the elements contained in the respective page and element actions/flow.

The **Configurations** folder contains test automation config files with access to environment variables, connections, and security keys/credentials.

Advantages of Folder Structure

The advantages of the Folder structure are as follows:

- **Code Readability and Reusability:** Code readability is improved as the folders are organized to separate different files. It makes it easier for any new team member to easily understand the test flow. Additionally, the code can be reused across multiple projects without having to rewrite.
- **Simplifies collaboration:** Team contributions consistency is maintained to follow the same test automation approach.
- **Easier Updates:** Folder structure and page object model combined enables the test member to update the code more easily in case of any changes to features.
- **Improves code quality and delivery:** With folder structure, the code quality, and test runs are improved and help in easier delivery of code with a separate folder for reports and logs.

Steps for Page Object Model (POM)

The steps for creation of POM are as follows:

1. **Identify Application Pages:** The first step is to identify the different pages of the application. Each application page is required to have a separate page object class with all the elements on the page and element actions. For example, the login page has these basic elements a textbox for email and password, submit/sign in, sign up, and forgot password link.
2. **Define the Page Object Classes:** In the following example, for a sample e-commerce application, we have created Login Page, Home Page, Men Page, and Base Page. These separate classes in the POM are used by the test code to identify and interact with elements of the respective page without worrying about the underlying implementation. The page class files in the **Pages** folder are shown in *Figure 10.2*:

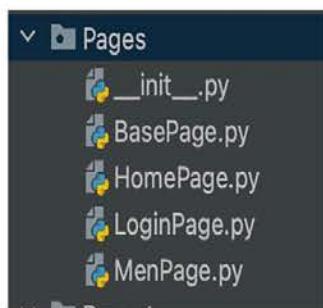


Figure 10.2: Application Pages

3. **Implementation of Page Object Classes:** Once the page object classes have been defined, the implementation needs to be done where the locator of all the elements of the page and actions performed on them are implemented. The **Base Page** serves as the foundation of all the other page classes. The base page class provides a centralized location of the basic functionality, that is common across all the web pages of the application. This includes functionalities such as clicking, typing text, waiting for elements, finding element status, and so on. All the page object class inherits from the Base Page class. The Base Page helps reduce duplication more maintainable and further organizes the code. If any changes are required to the basic functionality, we only need to update the base page class and the changes will be reflected across all the pages that inherit from it. In *Figure 10.2*, we saw all the Page Object Classes. Here is a brief description of each:
 - a. The **TestData.py** is present in the Configuration folder in the folder structure. In this example, it is used to store URLs and credentials.
 - b. The **BasePage.py** has the constructor which takes the driver instance. As described before, **BasePage.py** is the host for application automation base functionality, that will be used across all the page object classes. In the following example, it contains the function "**textType_action**" which takes the element and text as the argument and type the provided text in the element. The function also implements explicit wait with "**visibility_of_element_located**" as the expected condition, so that any page object class when inheriting the BasePage, does not require adding an explicit wait condition. Depending on the application, additional wait condition can be added.
 - c. The **LoginPage.py** has all the elements in the login page and login function along with click actions implemented using the inherited BasePage class.
 - d. Similarly, **HomePage.py** and **MenPage.py** have elements and element actions on the home page and Men's category page.

CODE – TestData.py

```
1. class TestData:  
2.     URL = "https://magento.softwaretestingboard.com/"  
3.     USERNAME = "admin12@gmail.com"  
4.     PASSWORD = "admin12!@"
```

CODE – BasePage.py

```
1. from selenium.webdriver.support import expected_conditions as ec  
2. from selenium.webdriver.support.wait import WebDriverWait  
3.  
4.  
5. class BasePage:  
6.  
7.     def __init__(self, driver):  
8.         self.driver = driver  
9.  
10.    def typeText_action(self, element, textType):  
11.        WebDriverWait(self.driver, 30).until(ec.visibility_of_  
12.            element_located(element)).send_keys(textType)  
13.    def click_action(self, element):  
14.        WebDriverWait(self.driver, 30).until(ec.visibility_of_  
15.            element_located(element)).click()  
16.    def get_elementText(self, element):  
17.        return WebDriverWait(self.driver, 30).until(ec.  
18.            visibility_of_element_located(element)).text()  
19.    def elementEnabledStatus(self, element):  
20.        return WebDriverWait(self.driver, 30).until(ec.  
21.            visibility_of_element_located(element)).is_enabled()  
22.    def elementDisplayedStatus(self, element):
```



```
23.         return WebDriverWait(self.driver, 30).until(ec.
24.             visibility_of_element_located(element)).is_displayed()
25.
26.     def get_pageTitle(self):
27.         return self.driver.title
```

CODE – LoginPage.py

```
1.  from selenium.webdriver.common.by import By
2.  from Pages.BasePage import BasePage
3.
4.
5.  class Login(BasePage):
6.      textbox_username_id = (By.ID, "email")
7.      textbox_password_id = (By.ID, "pass")
8.      button_signIn_id = (By.ID, "send2")
9.      forgotpassword_linktext = (By.LINK_TEXT, "Forgot Your
10.        Password?")
11.     createAccount_xpath = (By.XPATH, "//a[@class='action create
12.        primary']/span")
13.     customerLoginText = (By.XPATH, "//*[text()='Customer Login']")
14.
15.
16.     def loginToApp(self, username, password):
17.         BasePage.typeText_action(self, element=self.textbox_
18.             username_id, textType=username)
19.         BasePage.typeText_action(self, element=self.textbox_
20.             password_id, textType=password)
21.         BasePage.click_action(self, element=self.button_signIn_id)
22.     def clickForgotPassword(self):
23.         BasePage.click_action(self, element=self.forgotpassword_
24.             linktext)
```



```
23.  
24.     def clickCreateAccount(self):  
25.         BasePage.click_action(self, element=self.createAccount_  
26.             xpath)  
27.     def customerLoginTextDisplayed(self):  
28.         BasePage.elementDisplayedStatus(self, element=self.  
             customerLoginText)
```

CODE – HomePage.py

```
1. from selenium.webdriver.common.by import By  
2. from Pages.BasePage import BasePage  
3.  
4.  
5. class Home(BasePage):  
6.     signIn_link_loc = (By.LINK_TEXT, "Sign In")  
7.     MenCategoryItemLink = (By.XPATH, "//a[@role = 'menuitem']//  
8.         span[text()='Men'])"  
9.     def __init__(self, driver):  
10.        super().__init__(driver)  
11.  
12.    def clickSignIn(self):  
13.        BasePage.click_action(self, element=self.signIn_link_loc)  
14.  
15.    def clickMenCategoryItem(self):  
16.        BasePage.click_action(self, element=self.  
            MenCategoryItemLink)
```

CODE – MenPage.py

```
1. from selenium.webdriver.common.by import By  
2. from Pages.BasePage import BasePage  
3.  
4.
```



```
5. class MenPage(BasePage):
6.     mens_pants_category_loc = (By.XPATH, "//a[text()='Pants']")
7.     mens_pants_firstProduct_loc = (By.XPATH, "//div[@
8.         class='product-item-info']/parent::li/parent::ol/li[1]")
9.     mens_pants_size = (By.XPATH, "//span[text()='Size']/
10.         following-sibling::div/div[1]")
11.    mens_pants_color = (By.XPATH, "//span[text()='Color']/
12.         following-sibling::div/div[1]")
13.    addToCart = (By.ID, "product-addtocart-button")
14.    cartIcon = (By.XPATH, "/div[@data-block='minicart']")
15.    checkout_proceed = (By.ID, "top-cart-btn-checkout")
16.
17.    def __init__(self, driver):
18.        super().__init__(driver)
19.
20.    def clickMensPantsCategory(self):
21.        BasePage.click_action(self, element=self.mens_pants_
22.            category_loc)
23.
24.    def addMenPants_first_ToCart_checkout(self):
25.        BasePage.click_action(self, element=self.mens_pants_
26.            category_loc)
27.        BasePage.click_action(self, element=self.mens_pants_
28.            firstProduct_loc)
29.        BasePage.click_action(self, element=self.mens_pants_size)
30.        BasePage.click_action(self, element=self.mens_pants_color)
31.        BasePage.click_action(self, element=self.addToCart)
32.        BasePage.click_action(self, element=self.cartIcon)
33.        BasePage.click_action(self, element=self.checkout_proceed)
```

5. **Using the Page Object Class and Performing the Test:** Now that we have defined the page object classes, the next step is to write the test cases that make use of these page object classes to interact with the application. The test files in the Tests folder are shown in *Figure 10.3*. Each test is executed using the command “`pytest -v -s Tests/test_login.py`” with the file

name replaced with the test to be executed. The **conftest.py** as discussed in the previous chapter is the special file that pytest looks for and automatically imports during test collection.

In the following code, **fixtureSetup** takes the param list with Chrome. We can pass Firefox and other intended browser along with Chrome, to execute the tests in all the provided browsers. The **fixtureSetup** returns the driver instance, which can be used across all the tests. Instead of “**return**” we can use the “**yield**” statement to ensure the driver is cleaned up after the tests finished executing. By defining fixtures like this, we can reuse setup login across multiple tests while keeping the test focused on specific functionality being tested. Apart from using it for the driver, we can use the **conftest.py** to include the function to navigate to the home page, return different application pages, and so on.

Refer to the following *Figure 10.3*:

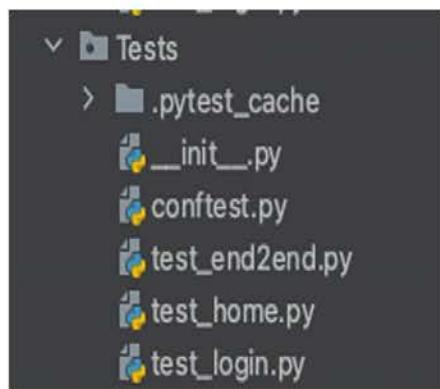


Figure 10.3: Application Tests

CODE – conftest.py

```

1. import pytest
2. from selenium import webdriver
3. from webdriver_manager.chrome import ChromeDriverManager
4. from webdriver_manager.firefox import GeckoDriverManager
5. from selenium.webdriver.chrome.service import Service
6.
7.
8. @pytest.fixture(params=['Chrome'], scope="class")
9. def fixtureSetup(request):
10.     driver = None

```

```
11.     if request.param == "Chrome":  
12.         driver = webdriver.Chrome(service=Service(ChromeDriver-  
13.             Manager("").install()))  
14.     elif request.param == "Firefox":  
15.         driver = webdriver.Firefox(service=Service(GeckoDriver-  
16.             Manager("").install()))  
17.     return driver
```

CODE – test_login.py

```
1. import time  
2. from selenium.webdriver.common.by import By  
3. from Pages.LoginPage import Login  
4. from Pages.HomePage import Home  
5. from Configurations.TestData import TestData  
6.  
7.  
8. class Test_Login():  
9.  
10.    def test_loginPageTitle(self, fixtureSetup):  
11.        self.driver = fixtureSetup  
12.        self.driver.get(TestData.URL)  
13.        Home(self.driver).clickSignIn()  
14.        lp = Login(self.driver)  
15.        pageTitle = lp.get_pageTitle()  
16.        assert "Customer Login" in pageTitle, "Not a valid Page"  
17.  
18.    def test_successful_login(self, fixtureSetup):  
19.        self.driver = fixtureSetup  
20.        self.driver.get(TestData.URL)  
21.        self.driver.find_element(By.LINK_TEXT, "Sign In").click()  
22.        lp = Login(self.driver)  
23.        lp.loginToApp(username=TestData.USERNAME,  
24.                        password=TestData.PASSWORD)
```

```

24.     time.sleep(2)
25.     pageTitle = lp.get_pageTitle()
26.     assert "website to practice selenium" in pageTitle,
27.           "Login Not Successful"
28.     def test_invalid_login(self, fixtureSetup):
29.         self.driver = fixtureSetup
30.         self.driver.get(TestData.URL)
31.         self.driver.find_element(By.LINK_TEXT, "Sign In").click()
32.         lp = Login(self.driver)
33.         lp.loginToApp(username="incorrectUserName",
34.                         password="incorrectPassword")
35.         time.sleep(2)
36.         pageTitle = lp.get_pageTitle()
37.         assert "Login" in pageTitle, "Login Not Successful"

```

OUTPUT – test_login.py

The output can be seen in the following *Figure 10.4*:

```

(venv) yogashivnathivanan@yogashivnathivanan-MacBook-Pro:~/PycharmProjects/POMFramework$ python -m pytest Tests/test_login.py
=====
test session starts =====
platform darwin -- Python 3.10.7, pytest-7.2.2, pluggy-1.8.0 -- /Users/yogashivnathivanan/PycharmProjects/POMFramework/venv/bin/python
cachedir: .pytest_cache
rootdir: /Users/yogashivnathivanan/PycharmProjects/POMFramework
collected 3 items

Tests/test_login.py::Test_Login::test_login[Chrome] PASSED
Tests/test_login.py::Test_Login::test_successful_login[Chrome] PASSED
Tests/test_login.py::Test_Login::test_Invalid_login[Chrome] PASSED

===== 3 passed in 25.96s =====

```

Figure 10.4: Output

CODE – test_home.py

```

1. from Pages.HomePage import Home
2. from Configurations.TestData import TestData
3.
4.
5. class Test_Home():
6.
7.     def test_navigationToMenCategory(self, fixtureSetup):

```

```

8.         self.driver = fixtureSetup
9.         self.driver.get(TestData.URL)
10.        hp = Home(self.driver)
11.        hp.clickMenCategoryItem()
12.        pageTitle = hp.get_pageTitle()
13.        assert "Men" in pageTitle, "Not a valid Page"

```

OUTPUT – test_home.py

The output can be seen in the following *Figure 10.5*:

```
(venv) yogashivam@yogashivam-MacBook-Pro:~/PycharmProjects/OMRFramework$ pytest -v -s Tests/test_home.py
=====
test session starts =====
platform darwin -- Python 3.10.7, pytest-7.2.2, pluggy-1.8.0 -- /Users/yogashivam@yogashivam-MacBook-Pro:~/PycharmProjects/OMRFramework/venv/bin/python
cachedir: pytest_cache
rootdir: /Users/yogashivam@yogashivam-MacBook-Pro:~/PycharmProjects/OMRFramework
collected 1 item

Tests/test_home.py::Test_Home::test_navigationToMenCategory[Chrome] PASSED

=====
1 passed in 0.24s
=====
(venv) yogashivam@yogashivam-MacBook-Pro:~/PycharmProjects/OMRFramework$
```

Figure 10.5: Output

CODE – test_end2end.py

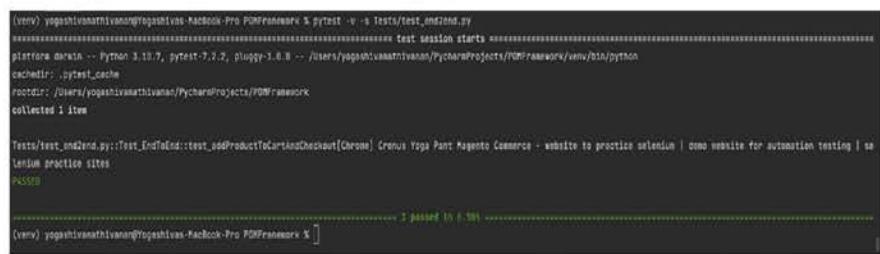
```

1. from Pages.HomePage import Home
2. from Pages.MenPage import MenPage
3. from Configurations.TestData import TestData
4.
5.
6. class Test_EndToEnd():
7.
8.     def test_addProductToCartAndCheckout(self, fixtureSetup):
9.         self.driver = fixtureSetup
10.        self.driver.get(TestData.URL)
11.        hp = Home(self.driver)
12.        hp.clickMenCategoryItem()
13.        mp = MenPage(self.driver)
14.        mp.addMenPants_first_ToCart_checkout()
15.        print(mp.get_pageTitle())

```

OUTPUT – test_end2end.py

The output can be seen in the following *Figure 10.6*:



```
(venv) yogeshivanan@yogeshivas-MacBook-Pro POMFramework % pytest -v -s Tests/test_end2end.py
=====
test session starts =====
platform darwin -- Python 3.10.7, pytest-7.2.2, pluggy-1.0.0 -- /users/yogeshivanan/PycharProjects/POMframework/venv/bin/python
cachedir: .pytest_cache
rootdir: /Users/yogeshivanan/PycharProjects/POMframework
collected 1 item

Tests/test_end2end.py::Test_EndToEnd::test_addProductToCartAndCheckout[Chrome] Cenus Yoga Pant Magento Commerce - website to practice selenium | demo website for automation testing | selenium practice sites
PASSED

=====
2 passed in 6.59s =====
(venv) yogeshivanan@yogeshivas-MacBook-Pro POMFramework %
```

Figure 10.6: Output

Best Practices for Implementing the Page Object Model

The following are some best practices for implementing the Page Object Model:

- **Identify the pages or features of the application:** The first step in implementing the POM is to identify the pages or features of the application that need to be tested. Each page or feature should have a separate Page Object class.
- **Define the Page Object classes:** Once the pages or features have been identified, the Page Object classes need to be defined. A Page Object class should encapsulate the functionality and elements of a single page or feature. The Page Object class should contain methods that perform actions on the elements of the page or feature and return data from the page or feature.
- **Implement the Page Object classes:** Once the Page Object classes have been defined, they need to be implemented. The implementation should include the elements of the page or feature, as well as the methods that perform actions on the elements.
- **Use the Page Object classes in the tests:** The Page Object classes can now be used in the tests. The tests should use the methods of the Page Object classes to interact with the application under the test.
- **Use the Page Factory pattern:** The Page Factory pattern is a design pattern that can be used to create Page Object classes. The Page Factory pattern uses annotations to identify the elements of the page and automatically initializes them when the Page Object is created. This makes it easier to create Page

Object classes and reduces the amount of boilerplate code that needs to be written.

- **Use a WebDriver abstraction layer:** A WebDriver abstraction layer is a set of classes that encapsulate the functionality of the WebDriver API. By using a WebDriver abstraction layer, tests become more maintainable and readable, as the details of the WebDriver API are hidden from the tests.
- **Use a Test Framework:** A Test Framework is a set of tools and libraries that can be used to create and run automated tests. Test Frameworks provide several benefits, including support for test case management, reporting, and parallel test execution.

Screenshots

We have seen how to create the framework folder structure to organize the code and leverage the page object model to make the framework robust. Now we will see the additional framework abilities that help in the automation implementation. Taking screenshots during the test execution is a common practice in test automation, as it can help in debugging and understanding the test failure. With Python Selenium and pytest, we can use the Selenium webdriver instance to take the screenshot and use the built-in pytest fixture ‘request’.

In the following code, we can use the WebDriver instance to take screenshots at any point during the test execution, but taking screenshots slows down the test execution. Thus, it is preferred to take screenshots only in case of failure. We used the “request” fixture to check if the test failed and only take screenshots then. It is best practice to use the test name as the screenshot name, so that it is easier to debug. Instead of hard-coding the name of the test as in the following example, we can use “`screenshot_path = "./screenshots/{}.png".format(request.node.name)"` to get the name of the failing test, or we can add a random number using the “random” package to attach a random number to uniquely identify the screenshots taken.

CODE – test_login.py

```
1. def test_loginPageTitle(self, fixtureSetup, request):
2.     self.driver = fixtureSetup
3.     self.driver.get(TestData.URL)
4.     Home(self.driver).clickSignIn()
```

```
5. #     self.driver.save_screenshot("./Users/yogashivamathivanan/
PycharmProjects/POMFramework/Screenshots/" + "test_
loginPageTitle.png")
6.     lp = Login(self.driver)
7.     pageTitle = lp.get_pageTitle()
```

```
8.         assert "Customer Login" in pageTitle, "Not a valid Page"
9.         # Take a screenshot if the test fails
10.        if request.node.rep_call.failed:
11.            screenshot_path = "./Screenshots/test_loginPageTitle.
png"
12.            self.driver.save_screenshot(screenshot_path)
```

Additionally, we have already discussed the “**pytest-html**” plugin which takes the screenshot and adds them to the report. Pytest-html generates a report file in the current working directory, but to save the report in a separate folder, use command “**pytest --html=path/to/report.html**”.

PyAutoGUI

PyAutoGUI is a Python module that provides a cross-platform **Graphical User Interface (GUI)** automation tool for Python. It allows control of mouse and keyboard operation along with GUI automation tasks.

Install the package using the command “**pip install pyautogui**”. In the following code, the mouse can be controlled using **moveTo()**, which moves the mouse cursor to the position (100, 100). **click()** clicks the left mouse button, **doubleClick()** double-clicks the left mouse button, and **rightClick()** right-clicks the mouse. **moveRel()** moves the mouse cursor relative to its current position, and **scroll()** scrolls the mouse up. To control the keyboard, **press()** presses a single key, and **typewrite()** types a string of characters.

Additionally, PyAutoGUI can also be used to take a screenshot of the entire screen using the function **screenshot()**. **screenshot(region)** takes a screenshot of the region defined in ‘region’ tuple. **save()** saves the screenshot to a file. In the following code, the screenshot is stored in the same directory. Note, in case of error related to image, “**NameError: name 'Image' is not defined**” is displayed.

CODE - dateTImeEx.py

```
1. import pyautogui
2. from PIL import Image
```

```
3.  
4. # Move the mouse to coordinates (x=100, y=100)  
5. pyautogui.moveTo(100, 100)  
6.
```

```
7. # Click the left mouse button  
8. pyautogui.click()  
9.  
10. # Double-click the left mouse button  
11. pyautogui.doubleClick()  
12.  
13. # Right-click the mouse  
14. pyautogui.rightClick()  
15.  
16. # Move the mouse relative to its current position  
17. pyautogui.moveRel(50, 0)  
18.  
19. # Scroll the mouse up  
20. pyautogui.scroll(10)  
21.  
22. # Press the 'a' key  
23. pyautogui.press('a')  
24.  
25. # Type the string 'hello'  
26. pyautogui.typewrite('hello')  
27.  
28. # Press the 'enter' key  
29. pyautogui.press('enter')  
30.  
31. # Take a screenshot of the entire screen  
32. screenshot = pyautogui.screenshot()
```

```
33.  
34. # Save the screenshot to a file  
35. screenshot.save('screenshot.png')  
36.  
37. # Take a screenshot of a region of the screen  
38. region = (100, 100, 300, 300) # left, top, width, height
```

```
39. screenshot2 = pyautogui.screenshot(region=region)  
40.  
41. screenshot2.save('screenshot2.png')  
42.
```

While PyAutoGUI is a powerful tool for automating GUI tasks, there are some limitations to consider:

- **Speed:** PyAutoGUI may not be as fast as other automation tools due to the fact that it simulates mouse and keyboard actions. This can be an issue when automating tasks that require high-speed interactions or that need to be completed within a very short timeframe.
- **Positional accuracy:** PyAutoGUI relies on position-based interactions, which means that it performs actions based on the position of the mouse cursor. This can be a problem when the position of the GUI elements changes, as it may cause PyAutoGUI to fail to perform the intended action.
- **Platform compatibility:** PyAutoGUI is designed to work on multiple platforms, including Windows, macOS, and Linux. However, there may be some limitations or differences in functionality between platforms, which may affect the performance and reliability of PyAutoGUI.
- **Dependence on external factors:** PyAutoGUI relies on the external environment to function correctly, such as screen resolution and the availability of GUI elements. Changes to these external factors can impact the reliability and accuracy of PyAutoGUI.
- **Security:** PyAutoGUI can be used for malicious purposes, such as automating password cracking or other forms of unauthorized access. As such, it is important to use PyAutoGUI responsibly and to take appropriate measures to protect your system and data from unauthorized access.

Configurations

In the preceding example, we have used the “**TestData.py**” class within

configurations to read URLs, user credentials, and so on. The configuration folder structure can have files and modules related to the configuration of the application. Typically, a configuration folder might contain one or more configuration files such as .ini files, .json files, or .yaml files apart from .py files. The configuration data might include User credentials, Database connection settings, API authorization details, environment variables, logging settings, and debugging options. The configuration settings in a specific location help manage the configuration data for the project and easy modifications to the application behavior.

Configparser

We have used the .ini file earlier for adding markers, and we can use the .ini file to add the environment variables. A .ini file known as an initialization file is a configuration file format that contains application config info in key-value pairs. It is a simple readable file that allows users to easily modify configuration settings without modifying the source code. In Python, the “**configparser**” module works with the .ini file. It provides a structured way to read and write .ini files using dictionary-like syntax, thus making it easy to work with configuration settings in Python programs.

In the code, we first create a **configparser** object and call the method **read()**, to read the contents of the config.ini file. Then the **get()** method of configparser is used to retrieve the values of the parameters as string type added in the .ini file, which can be used in the test code.

CODE – config.ini

```
1. [credentials]
2. username = admin12@gmail.com
3. password = admin12!@
4.
5. [urls]
6. base_url = https://magento.softwaretestingboard.com/
7. login_url = https://magento.softwaretestingboard.com/customer/
   account/login/
8.
9. [DATABASE]
10. host = localhost
11. port = 5432
12. user = postgres
13. password = secret
```

CODE – readProperty.py

```
1. import configparser
2.
3. config = configparser.RawConfigParser()
4. config.read("./Configurations/config.ini")
5.
```

```

6. base_url = config.get('urls', 'base_url')
7. login_url = config.get('urls', 'login_url')
8.
9. username_qa = config.get('credentials', 'username')
10. password_qa = config.get('credentials', 'password')

```

CODE – test_login.py

```

1. from Pages.LoginPage import Login
2. from Utilities import readProperties
3.
4.
5. class Test_Login():
6.
7.     def test_loginPageTitle(self, fixtureSetup, request):
8.         self.driver = fixtureSetup
9.         self.driver.get(readProperties.login_url)
10.        lp = Login(self.driver)
11.        pageTitle = lp.get_pageTitle()
12.        assert "Customer Login" in pageTitle, "Not a valid Page"
13.

```

OS and Pathlib

The “os” and “pathlib” libraries are commonly used in test automation for directory manipulation operations, interaction with the file system, and operating system.

The ‘os’ module is a Python built-in library that provides a way to interact with the operating system. We can use the “os” module to access the file system, create, update & delete files and directories, and manipulate environment variables.

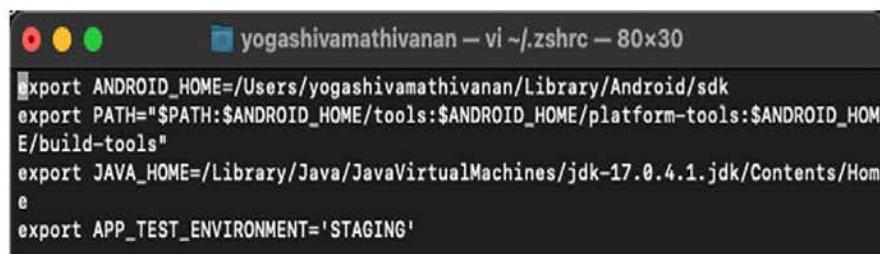
Let us look at some of the commonly used methods in the ‘os’ module:

- **os.environ:** A dictionary-like object that represents the environment variables of the current domain of the application. The module can be useful in test automation for the following reasons:
 - **Configuration Management:** In the previous code, we used the “`configparser`” module to retrieve the environment variables such as URLs, database connection, and credentials from the ‘.ini’ file. Test

automation often requires storing these configurations as environment variables and reading them using '`os.environ`'. This makes it easy to manage and update these values across different test environments.

- o **Data-driven testing:** In some high-priority sensitive instances, a test case is run multiple times with different data sets, storing the data as environment variables.
- o **CI/CD pipelines:** Continuous integration and deployment pipelines often use environment variables to store sensitive information like authorization keys and credentials. Using `os.environ` to access this information, ensures the test can run within the pipeline without hard coding the information in the script.

Let us add the environment variable to the environment variable "`APP_TEST_ENVIRONMENT`" as "`STAGING`" as shown in *Figure 10.7*, and it can be accessed as in the following code, using `os.environ` in line 8. Based on the value retrieved, the method "`getEnvUrlsCredentials`" returns the staging environment variable URL, username, and password, that can be used in the test shown in following test code. We can set a new environment variable as in line 25:



```
yogashivamathivanan - vi ~/.zshrc - 80x30
export ANDROID_HOME=/Users/yogashivamathivanan/Library/Android/sdk
export PATH="$PATH:$ANDROID_HOME/tools:$ANDROID_HOME/platform-tools:$ANDROID_HOME/build-tools"
export JAVA_HOME=/Library/Java/JavaVirtualMachines/jdk-17.0.4.1.jdk/Contents/Home
export APP_TEST_ENVIRONMENT='STAGING'
```

Figure 10.7: Setting Environment Variable

CODE – TestData

```
1. import os
2.
3.
4. class TestData:
5.     URL = "https://magento.softwaretestingboard.com/"
6.     USERNAME = "admin12@gmail.com"
7.     PASSWORD = "admin12!@"
8.     appEvn = os.environ.get("APP_TEST_ENVIRONMENT")
9.
```

```

10.     def getEnvUrlsCredentials(self):
11.         if self.appEnv == "STAGING":
12.             URL_STAGING = "https://magento.softwaretestingboard.
13.                         com/customer/account/login/"
14.             USERNAME_STAGING = "admin12@gmail.com"
15.             PASSWORD_STAGING = "admin12!@"
16.             return [URL_STAGING, USERNAME_STAGING, PASSWORD_STAGING]
17.         elif self.appEnv == "QA":
18.             URL_QA = "https://magento.qa.softwaretestingboard.com/"
19.             USERNAME_QA = "admin12_qa@gmail.com"
20.             PASSWORD_QA = "admin12!@"
21.             return [URL_QA, USERNAME_QA, PASSWORD_QA]
22.     def getCurrentUsersHomeDirectory(self):
23.         home_dir = os.environ['HOME']
24.         print(home_dir)
25.         os.environ['TEST_VAR'] = "Test_Value"
26.         print(os.environ['TEST_VAR'])
27.
28.
29. if __name__ == '__main__':
30.     TestData().getCurrentUsersHomeDirectory()

```

CODE – test_login.py

```

1. from Pages.LoginPage import Login
2. from Configurations.TestData import TestData
3.
4.
5. class Test_Login():
6.
7.     def test_loginPageTitle(self, fixtureSetup, request):
8.         self.driver = fixtureSetup
9.         self.driver.get(TestData()\getEnvUrlsCredentials()\qa\

```

```
10.     lp = Login(self.driver)
11.     pageTitle = lp.get_pageTitle()
12.     assert "Customer Login" in pageTitle, "Not a valid Page"
13.
```

Consider the following:

- **os.listdir():** This method returns a list of all files and directories in the specified path.
- **os.mkdir():** This method creates a new directory at the specified path.
- **os.rmdir():** This method deletes the specified directory.
- **os.rename():** This method renames the specified file or directory.
- **os.path.join():** This method is used to join two or more paths into a single path. It can be used to concatenate directory names and file names into a full path.
- **os.path.abspath():** This method returns the absolute path of a file or directory.
- **os.path.dirname():** This method returns the directory name of a path.
- **os.getcwd():** The method returns the current working directory as a string.

CODE – OSPathLib.py

```
1. import os
2.
3. #os.path.join()
4. path = os.path.join('/Users/yogashivamathivanan' + '/PycharmProjects/POMFramework', 'testfile.txt')
5. print(path)
6.
7. #os.listdir()
8. files = os.listdir('/Users/yogashivamathivanan/PycharmProjects/POMFramework')
9. print(files)
10.
11. #os.rename()
```

```

12. os.rename('/Users/yogashivamathivanan/PycharmProjects/POMFramework/TestData/test_login.py', '/Users/yogashivamathivanan/PycharmProjects/POMFramework/TestData/test2_login.py')

13.

14. #os.path.exists()

15. print(os.path.exists('/Users/yogashivamathivanan'))

16.

```

The ‘pathlib’ is like an extension to the ‘os’ library which was introduced in Python 3.4 as an alternative to the **os.path()** method. It provides an object-oriented interface for working with file paths and directories, making it easier to manipulate resource paths. Let us look at some of the commonly used methods in the **‘pathlib’** module:

- **Path():** This method is used to create a Path object, which represents a file or directory path.
- **Path.joinpath():** This method joins two or more paths together.
- **Path.home():** This method returns the user’s home directory as a ‘Path’ object.
- **Path.resolve():** This method returns the absolute path of a file or directory.
- **Path.parent:** This attribute returns the parent directory of a path.
- **Path.exists():** This method returns True if a file or directory exists, and False otherwise.
- **Path.mkdir() and Path.makedirs():** These methods are used to create a new directory. **.mkdir()** creates only one directory, while **.makedirs()** creates all intermediate directories if they do not exist.

Logging

Logging is a critical aspect of any testing and test automation framework. Logging helps captures the information about the execution of tests and provides insights into the behavior of the application being tested. Following are the significance and benefits of logging in to the test automation framework:

- **Debugging/Root Cause Analysis:** Logging contains the execution flow information, which helps in identifying the errors and root cause of the failure and pinpointing the module with the issue. Hence, logging helps to ~~resolve it quickly and prevent it from happening in the future~~.

- **Performance and Scalability:** Logging can be used as application performance monitoring, which helps in resolving bottlenecks and improving the system's performance.
- **Reporting and Collaboration:** Logging helps to generate detailed reports of executions and system failures and helps communicate the application behavior to the team in a more precise and detailed format.
- **Traceability:** Logging provides traceability, enabling tracking of test execution and the results obtained at each.
- **Compliance:** Logging is useful to track the application changes over time, often used for regulatory compliance.

Python logging

The logging module in Python is a built-in module that allows the generation of log messages from code. We can also use the module to write the logs to a specific log file, console, or to any other output stream. Here are some of the concepts of logging module in Python:

- **Loggers:** An object that is used to generate log messages. Each logger is identified by a name, and we can create multiple loggers in a Python program.
- **Handlers:** An object that is responsible for writing log messages to an output stream such as a file or console. Each logger can have multiple loggers attached to it.
- **Levels:** Log messages are classified into different levels based on severity.
- **Formatters:** An object that defines the format of the log messages. Formatter are attached to handlers and each handler can have a different formatter attached to it. The logging module has several built-in formatters or allows to create a custom formatter.
- **Filters:** An object that allows to selectively process log messages based on their attributes.

Logging level

The logging level provides a way to categorize the log messages based on their severity or importance, which can be useful for diagnosing and debugging errors.

severity or importance, which can be useful for diagnosing and debugging errors. By using the appropriate logging level for each log, we can ensure the right level of detail is captured in the logs and can quickly identify and address any issues. Here are the different logging levels and definitions as provided by Python, which can be referred at the Python documentation: <https://docs.python.org/3/library/logging.html>:

- **DEBUG:** Detailed information, typically of interest only when diagnosing problems.
- **INFO:** Confirmation that things are working as expected.
- **WARNING:** An indication that something unexpected happened or indicative of some problem in the near future (for example, ‘disk space low’). The software is still working as expected.
- **ERROR:** Due to a more serious problem, the software has not been able to perform some functions.
- **CRITICAL:** A very serious error, indicating that the program itself may be unable to continue running.

These logging levels are hierarchical. For example, setting the logging level to WARNING will result in all WARNING, ERROR, and CRITICAL messages being logged, but INFO and DEBUG messages will be ignored.

Logging Formatter

The formatter is attached to a handler and is responsible for formatting the log messages before they are output to the file or console. The syntax is:

```
"logging.Formatter('%(asctime)s %(name)s %(module)s %(funcName)s\n%(levelname)s %(message)s'):"
```

This formatter uses the following placeholders:

- **%(asctime)s:** The timestamp when the log message was created.
- **%(name)s:** The name of the logger that generated the log message.
- **%(module)s:** The name of the module where the log message was generated.
- **%(funcName)s:** The name of the function where the log message was generated.
- **%(levelname)s:** The severity level of the log message.
- **%(message)s:** The log message.

Output:

```
2022-03-10 12:34:56 name logger name module name function INFO This is
```

message.

In addition, “**datefmt**”, is an optional parameter in the logging. The Formatter that specifies the timestamp format, is used to format the timestamp provided by the ‘**asctime**’ placeholder in the format string passed to the ‘**Formatter**.’ If it is not specified, the default format for the timestamp is **%Y-%m-%d %H:%M:%S,%s**. The **datefmt**

parameter takes a string argument that specifies the format of the timestamp. The format string can contain any combination of the following codes:

- **%Y:** Year with century as a decimal number.
- **%m:** Month as a decimal number.
- **%d:** Day of the month as a decimal number.
- **%H:** Hour (24-hour clock) as a decimal number.
- **%M:** Minute as a decimal number.
- **%S:** Second as a decimal number.
- **%z:** UTC offset in the form +HHMM or -HHMM.
- **%Z:** Time zone name.

For example, if you want to format the timestamp as YYYY-MM-DD HH:MM:SS, you can use the following format string:

```
formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s -\n%(message)s', datefmt='%Y-%m-%d %H:%M:%S')
```

In the following Python code, to use the Python in-built logging module, simply import logging. Then logger is created with the name “**__name__**”, which is a special built-in variable that holds the name of the current module. By default, the logger is configured to write log messages to the console. Then set the log level to INFO, so that the DEBUG message will not be logged. Create the file handler to log the message to a separate log file. The formatter is created and set to the file handler and then added to the logger object created.

CODE – PyLogging.py

1. import logging
- 2.
3. # Create a Logger object
4. logger = logging.getLogger(__name__)
- 5.

```
6. # Set the Log Level to INFO  
7. logger.setLevel(logging.INFO)  
8.  
9. # Create a file handler  
10. file_handler = logging.FileHandler('../Logs/example.log')  
11.
```

```
12. # Create a formatter  
13. formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)s')  
14.  
15. # Set the formatter on the file handler  
16. file_handler.setFormatter(formatter)  
17.  
18. # Add the file handler to the Logger  
19. logger.addHandler(file_handler)  
20.  
21. # Log some messages  
22. logger.debug('This is a debug message')  
23. logger.info('This is an informational message')  
24. logger.warning('This is a warning message')  
25. logger.error('This is an error message')  
26. logger.critical('This is a critical message')
```

OUTPUT – example.log

1. 2023-03-10 22:05:32,036 - __main__ - INFO - This is an informational message
2. 2023-03-10 22:05:32,037 - __main__ - WARNING - This is a warning message
3. 2023-03-10 22:05:32,037 - __main__ - ERROR - This is an error message
4. 2023-03-10 22:05:32,037 - __main__ - CRITICAL - This is a critical message

Python logging in Framework

We can use the Python logging module to capture information about the test execution, to get more insight into the application behavior, and to identify errors. Let us utilize the folder structure and create “**LoggerInfo.py**” in the utility folder and write the logs in the Logs folder. Placing logs in separate folders helps in easy access to the logs, and easy maintenance ensures logs are organized and helps secure sensitive data.

The LoggerInfo utility class initializes the logging with basic configuration with name and location of the log file, log message format along with date and time format. The logger object is returned and utilized in the test. Within the test, the logger object is used to generate log messages at different points throughout the test execution and logs to the output file.

CODE – Utilities.LoggerInfo.py

```
1. import logging
2.
3.
4. class LogInfo:
5.     @staticmethod
6.     def logGen():
7.         logging.basicConfig(filename='../../Logs/test1234.log',
8.                             level=logging.DEBUG, format="(asctime)s: %(levelname)s:
9.                             %(message)s", datefmt="%m/%d/%Y %I:%M:%S %p")
10.    return logging.getLogger()
```

CODE - Test_login.py

```
1. from Pages.LoginPage import Login
2. from Configurations.TestData import TestData
3. from Utilities.LoggerInfo import LogInfo
4.
5.
6. class Test_Login():
7.     logger = LogInfo.logGen()
8.
```

```
9.     def test_logging(self, fixtureSetup):
10.        self.logger.info("_____test_loginPageTitle_____")
11.        self.driver = fixtureSetup
12.        self.logger.info("_____Fixture Setup Successfully
13.        created Webdriver instance_____")
14.        self.driver.get(TestData().getEnvUrlsCredentials()[0])
15.        self.logger.info("_____Application URL Successfully
16.        retrieved and logged in_____")
```

```
15.     lp = Login(self.driver)
16.     pageTitle = lp.get_pageTitle()
17.     self.logger.info("_____Login Page title Retrieved
18.     successfully_____")
19.     assert "Customer Login" in pageTitle, "Not a valid Page"
20.     self.logger.info("_____Assertion Passed_____")
```

OUTPUT – Logs.test.log

1. 03/07/2023 11:59:55 PM: INFO: _____test_loginPageTitle_____
2. 03/07/2023 11:59:55 PM: INFO: _____Fixture Setup Successfully
 created Webdriver instance_____
3. 03/07/2023 11:59:55 PM: INFO: _____Application URL Successfully
 retrieved and logged in_____
4. 03/07/2023 11:59:55 PM: INFO: _____Login Page title Retrieved
 successfully_____
5. 03/07/2023 11:59:55 PM: INFO: _____Assertion Passed_____

Parallel Test Execution

Parallel test execution is the process of running tests concurrently across multiple processors or machines, rather than sequentially on a single processor. This significantly reduces the execution time, especially in very large test suites. The total execution time is reduced by a factor equal to the number of processors or machines used. In the sequential test execution, each test must complete for the next test to run. Furthermore, parallel test execution can help to improve efficiency and optimize resource utilization in the testing process.

Pvtest-xdist

- J -----

Pytest-xdist is a plugin from pytest, that enables parallel execution by distributing tests across multiple processors or machines. **Pytest-xdist** uses a centralized scheduling mechanism to ensure that each test is executed only once and to gather the collective result of the test. It provides options to control the number of processors or machines to use for complete test execution.

Install **pytest-xdist** using the command “**pip install pytest-xdist**” and when executing the test, specify the number of processors to be used for parallel execution. In *Figure 10.8*, the two test processors are used by passing the ‘-n’ option to the pytest command, followed by the number of processors “**pytest -v -s -n 2 Tests/test_login.py**”.

test_login.py. Moreover, as in *Figure 10.8*, the **pytest-xdist** is scheduling the tests using LoadScheduling to control the execution.

Pytest-xdist can also run the test across multiple machines, which requires SSH access to remote machines and configuration file “**pytest.ini**” within the project to specify pytest options, logging information, and remote machine hostnames. **pytest-xdist** detects the file and distributes the load across the machines.

Refer to *Figure 10.8*:

```
(venv) yoganivaa@yoganivaa-MacBook-Pro PDMFramework % pytest -v -s -n 2 Tests/test_login.py
=====
test session starts =====
platform darwin -- Python 3.10.7, pytest-7.2.2, pluggy-1.8.0 -- /Users/yoganivaa@yoganivaa-MacBook-Pro/PycharmProjects/PDMFramework/venv/bin/python
cachedir: .pytest_cache
rootdir: /Users/yoganivaa@yoganivaa-MacBook-Pro/PycharmProjects/PDMFramework
plugins: adxit-3.2.0
[pid] darwin Python 3.10.7 cvidt /Users/yoganivaa@yoganivaa-MacBook-Pro/PycharmProjects/PDMFramework
[pid] darwin Python 3.10.7 cvidt /Users/yoganivaa@yoganivaa-MacBook-Pro/PycharmProjects/PDMFramework
[pid] Python 3.10.7 (v3.10.7:icecd13300, Sep 5 2022, 24:02:02) [Clang 13.0.0 (clang-13300.0.29.30)]
[pid] Python 3.10.7 (v3.10.7:icecd13300, Sep 5 2022, 24:02:02) [Clang 13.0.0 (clang-13300.0.29.30)]
[pid] + [pid] 14
scheduling tests via LoadScheduling

Tests/test_Login.py::Test_Login::test_successful_Login[Chrome]
Tests/test_Login.py::Test_Login::test_logging[Chrome]
[pid] PASSED Tests/test_Login.py::Test_Login::test_logging[Chrome]
Tests/test_Login.py::Test_Login::test_logpageTitle[Chrome]
[pid] FAILED Tests/test_Login.py::Test_Login::test_logpageTitle[Chrome]
[pid] + [pid] 14
[pid] - Downloading: 10.9MB [00:00, 8.0MB/s]

[pid] FAILED Tests/test_Login.py::Test_Login::test_successful_Login[Chrome]
[pid] - Downloading: 10.9MB [00:00, 8.0MB/s]

[pid] PASSED Tests/test_Login.py::Test_Login::test_logpageTitle[Firefox]
Tests/test_Login.py::Test_Login::test_successful_Login[Firefox]
[pid] FAILED Tests/test_Login.py::Test_Login::test_logging[Firefox]
[pid] PASSED Tests/test_Login.py::Test_Login::test_successful_Login[Firefox]
```

Figure 10.8: Parallel Test Execution using pytest-xdist

Data – Driven Testing with Data Source

In data-driven testing automation, the test scripts run with multiple sets of input data. The approach is used to automate multiple test cases by executing one test

The approach is used to automate multiple test cases by executing one test case with different data inputs, rather than creating separate test cases for different inputs, thus helping in increasing the test coverage, reusability, faster execution, and easier maintenance.

There are several ways to implement data-driven testing automation using different data sources including CSV files, Excel files, Databases, or API endpoints/JSON files. There are multiple Python modules that can help us interact and get data out from the data sources.

Here are the steps involved in data-driven testing:

- **Identify the data inputs:** Determine the test data that is required to test the application.

- **Define the test scenario:** Write test cases of different valid and invalid test scenarios, based on the data inputs and expected output.
- **Create the data source:** Store the test data in an external data source such as excel or csv. Establish database or API connection to retrieve the required test data.
- **Read the data source:** Once the data is retrieved or stored in the data source, write utilities to manipulate the test data. We can use Python libraries to achieve this.
- **Execute the test cases:** Use the retrieved data within tests and execute.
- **Validate the results:** Validate the expected results.

Openpyxl

Openpyxl is used to write and read data from Excel spreadsheet files. To use the library, install openpyxl using pip command “**pip install openpyxl**”.

In the following utility code, Openpyxl is used to write the utility function “**getRowCount**” to return the number of rows in the worksheet. It is important to include or exclude the header row, based on test requirements. Similarly, the function “**getColumnCount**” returns the number of columns, “**readData**” returns cell data, based on the passed sheetname, row, and column, and “**writeData**” writes the data into the cell based on the passed sheetname, row, and column and saves the workbook.

We have created data source Excel sheet in the location **testData/DDT-Login.xlsx** within the framework. In the test, we are calling the utility functions of **XLUtils.py** and retrieving the username and password data from the sheet to execute the same test with multiple sets of user credential data. Refer to Table 10.1.

CSV with multiple sets of user credential data. Refer to this tool.

Username	Password
admin12@gmail.com	admin12!@
admin23@gmail.com	admin23@#

Table 10.1: Excel Spreadsheet file Data

CODE – Utilities.XLUtils.py

```
1. import openpyxl  
2.  
3.  
4. def getCount(file, sheetName):
```

```
5.     workbook = openpyxl.load_workbook(file)
6.     sheet = workbook[sheetName]
7.     return sheet.max_row
8.
9.
10.    def getColumnCount(file, sheetName):
11.        workbook = openpyxl.load_workbook(file)
12.        sheet = workbook[sheetName]
13.        return sheet.max_column
14.
15.
16.    def readData(file, sheetName, rownum, columnnum):
17.        workbook = openpyxl.load_workbook(file)
18.        sheet = workbook[sheetName]
19.        return sheet.cell(row=rownum, column=columnnum).value
20.
21.
22.    def writeData(file, sheetName, rownum, columnnum, data):
23.        workbook = openpyxl.load_workbook(file)
24.        sheet = workbook[sheetName]
25.        sheet.cell(row=rownum, column=columnnum).value = data
26.        workbook.save(file)
```

CODE - Test_login.py

```
1. from Pages.LoginPage import Login
2. from Utilities import readProperties
3. from Utilities import XLUtils
4. from pathlib import Path
5.
6.
7. class Test_Login():
8.
9.     def test_successful_login_ddt(self, fixtureSetup):
```



```

10.     self.driver = fixtureSetup
11.     self.driver.get(readProperties.login_url)
12.     lp = Login(self.driver)
13.     path = Path(__file__).parent.parent.joinpath('TestData/
DDT-Login.xlsx')
14.     rowCount = XLUtils.getRowCount(path, 'Sheet1')
15.     for i in range(2, rowCount+1):
16.         lp.loginToApp(XLUtils.readData(path, 'Sheet1', i, 1),
XLUtils.readData(path, 'Sheet1', i, 2))
17.         pageTitle = lp.get_pageTitle()
18.         assert "My Account" in pageTitle, "Login Not Successful"
19.     self.driver.close()

```

CSV

The CSV package can be used to read and write data to and from CSV files, which is a more popular form of storing tabular data. We can use a similar approach of creating a utility file with functions and calling the function within the test. In the following code, we open the CSV file with username and password using the “open” function and pass the file path and the mode ‘r’ to indicate that we want to read the file. We then create a ‘**DictReader**’ object from the CSV package, which allows us to read the file as a dictionary, where the keys are column headers, and the values are the corresponding data in each row. We then iterate through each row in the file and extract the ‘username’ and ‘password’ values using the dictionary keys.

CSV File: CSV-Login.csv

```

username,password
admin12@gmail.com,admin12!@
admin23@gmail.com,admin23#@#

```

CODE - Test_login2.py

```

1. import csv
2. from Pages.LoginPage import Login
3. from Utilities import readProperties
4. from pathlib import Path
5.

```

```
6.  
7. class Test_Login2():  
8.  
9.     def test_successful_login_ddt(self, fixtureSetup):  
10.        self.driver = fixtureSetup  
11.        self.driver.get(readProperties.login_url)  
12.        lp = Login(self.driver)  
13.        path = Path(__file__).parent.parent.joinpath('TestData/  
CSV-Login.csv')  
14.        with open(path, 'r') as csvfile:  
15.            reader = csv.DictReader(csvfile)  
16.            for row in reader:  
17.                username = row['username']  
18.                password = row['password']  
19.                lp.loginToApp(username, password)  
20.                pageTitle = lp.get_pageTitle()  
21.                assert "My Account" in pageTitle, "Login Not  
Successful"  
22.  
23.        self.driver.close()
```

Pandas

Pandas is a most popular widely used data manipulation library of Python. It provides various functions to read and work with structured data, such as spreadsheets, SQL tables, or CSV files. Pandas is built on top of the NumPy library. Pandas module is dominantly used in data science and is particularly useful for tasks such as data cleaning, data transformation, data aggregation, and data visualization. The abilities of the pandas library features are too vast and will require a chapter or book of its own. Let us understand some basics of the pandas library and how it can be used for data-driven testing automation.

The pandas code base can be found in the corresponding GitHub library. Install the pandas library using the pip command “**pip install pandas**”. Following are a few pandas concepts:

- **Pandas Series:** A Pandas series is a one-dimensional array that can hold data of any data type. It is like a Python list with more functionality and flexibility

A Pandas series has two main components:

- o **Index:** Pandas generate a default `index` consisting of integers from 0, but can be customized by passing a list of labels or using key-value objects when creating the series as in the following code example.
- o **Value**
- **Pandas Dataframe:** A pandas dataframe is a two-dimensional labeled data structure that can hold data of different data types. It is like a spreadsheet or SQL table with rows and columns of data. Dataframe also has an index component which can also be customized.

CODE - PandasLearn.py

```
1. import pandas as pd
2.
3. print("PANDAS SERIES")
4. x = [3, 12, 36]
5. y = {"Fruit1": "apple", "Fruit2": "banana", "Fruit3": "Orange"}
6. index = ['a', 'b', 'c']
7. var_x = pd.Series(x, index)
8. var_y = pd.Series(y)
9. print(var_y)
10.
11.
12. print("PANDAS DATAFRAMES")
13. animal_data = {
14.     'animals': ['dogs', 'cats', 'rats', 'camels'],
15.     'name': ['Alice', 'Bob', 'Charlie', 'Adam'],
16.     'age': [2, 4, 7, 9],
17.     'city': ['New York', 'Paris', 'London', 'Tokyo']
18. }
19. index = ['a', 'b', 'c', 'd']
20. var_animal = pd.DataFrame(animal_data, index)
21. print(var_animal)
```

OUTPUT

The output can be seen in the following *Figure 10.9*:

PANDAS SERIES				
Fruit1	apple			
Fruit2	banana			
Fruit3	Orange			
dtype: object				
PANDAS DATAFRAMES				
	animals	name	age	city
a	dogs	Alice	2	New York
b	cats	Bob	4	Paris
c	rats	Charlie	7	London
d	camels	Adam	9	Tokyo

Figure 10.9: Output

Pandas can be used for data-driven testing such as openpyxl and csv libraries, although in more advanced functionality. Pandas is not limited to one data source and can work with CSV, excel, databases, JSON, and so on. In the following code, pandas can read the test data in CSV file and store it in a Pandas dataframe and iterate through each row in the Dataframe using the `iterrows()` function to extract the credential data.

CODE - PandasLearn.py

```

1. import pandas as pd
2. from Pages.LoginPage import Login
3. from Utilities import readProperties
4. from pathlib import Path
5.
6.
7. class Test_Login2():
8.
9.     def test_successful_login_ddt_pandas(self, fixtureSetup):
10.         self.driver = fixtureSetup
11.         self.driver.get(readProperties.login_url)

```

```
12.         lp = Login(self.driver)
```

```
13.         path = Path(__file__).parent.parent.joinpath('TestData/  
CSV-Login.csv')  
14.         test_data = pd.read_csv(path)  
15.         for index, row in test_data.iterrows():  
16.             username = row['username']  
17.             password = row['password']  
18.             lp.loginToApp(username, password)  
19.             pageTitle = lp.get_pageTitle()  
20.             assert "My Account" in pageTitle, "Login Not Successful"  
21.  
22.         self.driver.close()
```

NumPy

NumPy is a Python library for performing numerical computations with large, multi-dimensional arrays and matrices. It provides vector and matrix operations along with mathematical functions. It is dominantly used for applications with numerical computations with large arrays, image processing, signal processing, and scientific simulations.

NumPy can be used in test automation in several ways:

- **Data generation:** Mostly, NumPy is used to generate test data for the advanced application. For example, to generate test images from testing an image processing algorithm.
- **Data manipulation:** NumPy can manipulate data in many ways, such as sorting, filtering, and transforming data.
- **Mathematical operations:** NumPy can be used to perform mathematical operations on test data.
- **Statistical analysis:** Statistical analysis such as mean, standard deviation, and correlation of dataset can be accomplished using NumPy.

Pickle

'Pickle' is a Python serialization and de-serialization module that converts complex

Python objects into byte streams and back. It allows the user to store the object in a file or transfer them over the network and then restore them to their original state.

Pickle library can be used to serialize and de-serialize a wide range of Python objects, including lists, dictionaries, sets, and even user-defined objects.

In the following code, we define a list of dictionaries 'employee' containing names and ages. We use '`pickle.dump()`' to serialize and save it to a file '`people.pickle`' which is generated in the same directory in the following example. This encoded file can be transferred as per need. To deserialize the list of dictionaries from the file and assign it to a variable '`loaded_employee`' and print it. We can also assign the byte string to a variable '`serialized_employee`' and the output is byte string representation of the serialized data.

Note: If we want to save the serialized data to a file, it is recommended to use binary mode '`wb`' or '`rb`'.

CODE - PandasLearn.py

```
1. import pickle
2.
3. employee = [
4.     {'name': 'Alice', 'age': 25},
5.     {'name': 'Bob', 'age': 30},
6.     {'name': 'Charlie', 'age': 35}
7. ]
8.
9. # Serialize the list of dictionaries
10.with open('people.pickle', 'wb') as f:
11.    pickle.dump(employee, f)
12.
13.serialized_data = pickle.dumps(employee)
14.print(serialized_data)
15.
16.
17.# Deserialize the list of dictionaries
```

```

18. with open('people.pickle', 'rb') as f:
19.     loaded_employee = pickle.load(f)
20.
21. print(loaded_employee)

```

OUTPUT

The output can be seen in the following *Figure 10.10*:

```

b'\x80\x94\x95D\x00\x00\x00\x00\x00\x00\x00\x00\x00\x94(\x94\x8c\x04name\x94\x8c\x05Alice\x94\x8c\x03age\x94K\x19u\x
[{'name': 'Alice', 'age': 25}, {'name': 'Bob', 'age': 30}, {'name': 'Charlie', 'age': 35}]

Process finished with exit code 0

```

Figure 10.10: Output

Date Time

One of the most commonly used Python libraries is ‘`datetime`’. It provides classes to work with dates and times. It includes classes for representing dates, times and `timedeltas`, as well as functions for formatting and parsing dates and times. The various classes are:

- **`datetime.datetime`**: The class represents a specific date and time. It has several attributes, including year, month, day, hour, minute, second, and microsecond.
- **`datetime.date`**: The class represents a specific date, without any time information. It has attributes for year, month, and day.
- **`datetime.time`**: The class represents a specific time, without any date information with attributes for hour, minute, second, and microsecond.
- **`datetime.timedelta`**: The class represents a duration of time, which can be used for arithmetic with ‘`datetime`’ objects.

The `datetime` library can be used for formatting and parsing dates using the `'strftime()'` function.

Additionally, we can compare ‘`datetime`’, ‘`date`’, and ‘`time`’ objects using standard comparison operators such as ‘`<`’, ‘`>=`’, ‘`!=`’ and so on. Moreover, we can add or subtract a ‘`timedelta`’ object from a ‘`datetime`’ object, to get a new ‘`datetime`’ that is shifted by the specified amount of time.

The format string can contain various format codes that represent different parts of the `datetime` object. Here are some common format codes:

- `%Y`: 4-digit year
- `%m`: 2-digit month (01-12)
- `%d`: 2-digit day of the month (01-31)
- `%H`: 24-hour hour (00-23)

- `%M`: minute (00-59)
- `%S`: second (00-59)
- `%A`: full weekday name
- `%B`: full month name

CODE - dateTImeEx.py

```
1. import datetime
2.
3. dt = datetime.datetime(2023, 1, 1, 9, 0, 0)
4. d = datetime.date(2023, 1, 1)
5. t = datetime.time(9, 0, 0)
6. td = datetime.timedelta(days=1)
7. formatted = dt.strftime('%m/%d/%Y %H-%M-%S')
8.
9. print(dt)
10. print(d)
11. print(t)
12. print(td)
13. print(formatted)
14.
15. dt1 = datetime.datetime(2023, 1, 1, 9, 0, 0)
16. td1 = datetime.timedelta(hours=1, days=5)
17.
18. dt2 = dt1 + td1
19. print(dt2)
```

OUTPUT

```
1 2023-01-01 09:00:00
```

```
1. 2023-01-01 09:00:00
2. 2023-01-01
3. 09:00:00
4. 1 day, 0:00:00
5. 01/01/2023 09-00-00
6. 2023-01-06 10:00:00
```

Random

The ‘random’ Python package is also commonly used to generate random numbers and sequences, which can be used in test automation to generate unique test data in every test run.

In the following code, the ‘random’ package is used to generate a random integer number between 0 and 1 using ‘`randint()`’. Similarly, random float number between 0 and 1 using the `uniform()` function. A random element can be generated from a sequence (for example, a list or tuple), using the `choice()` function, and the `shuffle()` function can be used to shuffle a sequence.

CODE - dateTImeEx.py

```
1. import random
2.
3. random_int = random.randint(1, 10)
4. print(random_int)
5.
6. random_float = random.uniform(0, 1)
7. print(random_float)
8.
9. colors = ['red', 'green', 'blue']
10. random_color = random.choice(colors)
11. print(random_color)
12.
13. numbers = [1, 2, 3, 4, 5]
14. random.shuffle(numbers)
15. print(numbers)
16.
```

OUTPUT

1. 10
2. 0.7874930404269722
3. blue
4. [3, 1, 4, 5, 2]

Faker

Faker is used to generating fake data such as names, addresses, phone numbers, dates, and more. It can also be used to generate fake random test data. Install faker using the command “`pip install faker`”. Once installed, we need to create an instance of the faker class and then we can use the functions.

CODE - fakerLearn.py

```
1. from faker import Faker
2. import datetime
3.
4. fake = Faker()
5.
6. name = fake.name()
7. print(name)
8.
9. address = fake.address()
10. print(address)
11.
12.
13. phone_number = fake.phone_number()
14. print(phone_number)
15.
16. dateFake = fake.date_time()
17. print(dateFake)
18.
```

```
19. start_date = datetime.datetime.strptime('01-01-2022', '%d-%m-%Y').  
    date()  
20. end_date = datetime.datetime.strptime('31-12-2023', '%d-%m-%Y').  
    date()  
21.  
22. fake_date = fake.date_between_dates(date_start=start_date, date_  
    end=end_date)  
23.
```

```
24. print(fake_date)
```

OUTPUT

1. James Adams
2. 123 Judith Alley Apt. 048
3. Morristown, KY 21982
4. 920.857.4433x753
5. 2005-01-13 17:53:55
6. 2023-06-26

Conclusion

The automation framework is chosen based on testing requirements, language support, integration with application tools, maintenance, cost, and so on. However, a structure of the framework is common within any framework. To begin with, we should follow a folder structure for different components of test automation requirements such as configurations, test data, utilities, pages, and test automation outputs such as screenshots, logs, and reports. Within pages, we are following the POM which is a very common best practice for UI test automation. The POM helps in organizing the framework objects to improve maintaining and updating the framework.

In UI Automation, it is important to capture pages at various points during execution for debugging, or for the report. We use the 'Screenshot' and 'PyAutoGUI' modules of Python to take screenshots using the code. PyAutoGUI can also be used for mouse and keyboard interactions. The configuration involves providing all the test data required to start the application, which is sensitive information and needs to be stored in config files. We can use 'os' module to interact with config files to get the required test environment values and also use it alongside pathlib to work with files

and directories.

The ‘logging’ module helps to capture the log at different instances of test execution at different logging levels. Then we looked at the parallel execution of tests using the ‘`pytest-xdist`’ module and also one of the most significant steps in test automation, that is, data-driven testing and how to work with different data sources using the ‘`openpyxl`’, ‘`csv`’, and ‘`pandas`’ modules. Then, more Python modules can be very useful in different complex test cases such as ‘`numpy`’, ‘`random`’, and ‘`faker`’ to generate test data, `DateTime` to work with date time, and ‘`pickle`’ for serializing and deserializing.

Key facts

- The Page Object model is a design pattern in test automation that aims to improve test code readability and maintainability. The POM separates the classes for each application page and encapsulates the respective page elements and actions performed on them. This helps in code duplication.
- A well-structured folder hierarchy in test automation helps organize test files, resources, and configurations in a logical and consistent manner. This helps to easily locate and maintain test files and collaborate with team members.
- Configuration in test automation is used to store organization config files such as property files for test settings such as test environments, authorization keys, and so on.
- The ‘`screenshot`’ Python module is a simple way to capture screenshots of test execution screens. It is useful to create visual records of test cases or capturing errors during test execution.
- The ‘`logging`’ Python module is used to log messages of different log levels from the program, such as debug messages, informational messages, warning messages, and errors. It is widely used to help diagnose and troubleshoot issues in the code.
- The ‘`pytest-xdist`’ module is used for the parallel execution of tests across multiple processors and hosts. It helps in speeding up the execution, significant particularly for large test suites.
- Data-driven testing is a testing approach using an external data source, such as excel, CSV, or databases. Openpyxl is used to work with excel files, CSV is used to work with CSV files and pandas is a powerful data analysis library that provides tools to manipulate and read various data formats.

Questions

1. What is the significance and advantages of the page object model and folder structure?
2. What is the configuration folder used for?
3. How to read environment values from the config file?
4. How to transverse to the parent folder and join paths using the pathlib library?
5. What is the significance of logging and what are the different levels of logging?
6. How to take screenshots and save to a specific folder during test execution?

7. What is the command and option to run the tests in parallel using the 'pytest-xdist' library?
8. What is data-driven testing, and how to read row and column data using openpyxl, csv, and pandas modules?
9. How to generate fake names and addresses using the faker module?
10. What is a datetime Python module and how to generate a date of a specific format?
11. What is a 'pickle' module and how can it be used to serialize Python objects?

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



CHAPTER 11

Mobile Automation

Framework

Introduction

In the preceding chapters, we explored the significance of frameworks in automation. We learned how to implement the **Page Object Model (POM)** and folder structure to optimize and organize the automation framework for web UI application

automation. Now, we can apply a similar approach to create a framework structure for mobile automation using Appium. By leveraging the Appium concepts that we have learned so far and incorporating them into our framework, we can learn about all the benefits of a robust and efficient framework for mobile application automation. As we have already discussed in the earlier chapters, mobile application demand is on the rise, and new features and updates are more frequent in mobile applications compared to web applications. In such an environment, maintaining the mobile automation script within the framework is crucial in ensuring the quality of the application and keeping up with the high demand pace. We will learn how to set up the framework, create test cases, and run them against the mobile application.

Reporting is an essential aspect of automation testing, as it provides valuable insights into the test results and helps us to identify and troubleshoot issues. In the previous chapter, we learned about the pytest html report. In this chapter, let us

understand the Allure reporting tool, which provides a user-friendly interface for viewing detailed test results, including information on failed tests and test execution time.

Structure

In this chapter, we will discuss the following topics:

- Allure Reporting tool
 - Installation
 - Decorators in Allure reporting
 - Screenshot with Allure Report
 - Executing Test and Opening Allure Report
- Designing Appium Framework
 - Folder Structure
- Implementation of Appium Framework
 - Driver Initialization
 - Base Page and Screenshots
 - Configurations and Utilities
 - Pages

- o Tests and Execution
 - Execution of tests and output
- o Troubleshooting

Objectives

We will continue the framework discussed in this chapter; we will see how to leverage the framework advantages for mobile automation using Appium. The objective remains the same, but the concentration will be on mobile applications. We will start the discussion by learning about another dominantly used reporting tool, that is, the Allure reporting tool, which provides additional features with more detailed information on failed tests and test execution time. Then, we will create the automation framework from scratch and make the automation process efficient and maintainable. We will design and implement the folder structure and POM for mobile applications. We will also create base pages, and create and execute basic test cases.

Allure reporting tool

Allure, as mentioned, is one of the dominantly used powerful and flexible reporting tools that provides detailed and informative reports for automation test results. It was originally designed for Java-based testing frameworks such as TestNG and Junit, but now supports a wide range of testing frameworks, including Python. Allure reporting provides an easy-to-use web-based interface, to overview the test execution results including screenshots, logs, and error messages, which help in quickly identifying and troubleshooting issues.

Installation

Install the Allure reporting tool using the pip command “`pip install allure-pytest`”. We also need to set the environment variable using the `allure.bat` file. In Mac, we can install Allure using the Homebrew command “`brew installs allure`”.

For manual installation instructions, download the zip archive file from maven central <https://repo.maven.apache.org/maven2/io/qameta/allure/allure-commandline/2.9.0/>.

For Mac open terminal, type “`vi ~/.zshrc`” and add environment variable “`export PATH=$PATH:$HOME/allure-2.9.0/bin`”. Verify that Allure is installed using command “`allure --version`” as shown in the following *Figure 11.1*:

The image shows two terminal windows side-by-side. The top window is titled "yogashivamathivanan - vi ~/.zshrc - 113x42" and contains the following shell script code:

```
export ANDROID_HOME=/Users/yogashivamathivanan/Library/Android/sdk
export PATH="$PATH:$ANDROID_HOME/tools:$ANDROID_HOME/platform-tools:$ANDROID_HOME/build-tools"
export JAVA_HOME=/Library/Java/JavaVirtualMachines/jdk-17.0.4.1.jdk/Contents/Home
export APP_TEST_ENVIRONMENT='STAGING'
export PATH="$PATH:$HOME/allure-2.9.0/bin"
```

The bottom window is titled "yogashivamathivanan -- zsh - 113x42" and shows the output of the command "allure --version":

```
yogashivamathivanan@Yogashivas-MacBook-Pro ~ % allure --version
2.9.0
yogashivamathivanan@Yogashivas-MacBook-Pro ~ %
```

Figure 11.1: Allure Mac Installation – Adding Environment Variable and validate

For a Windows user, once the `allure.bat` file is extracted, navigate to the system environment variable and add the path, for example “`C:\allure2.9.0\bin`” and save as shown in *Figure 11.2*:

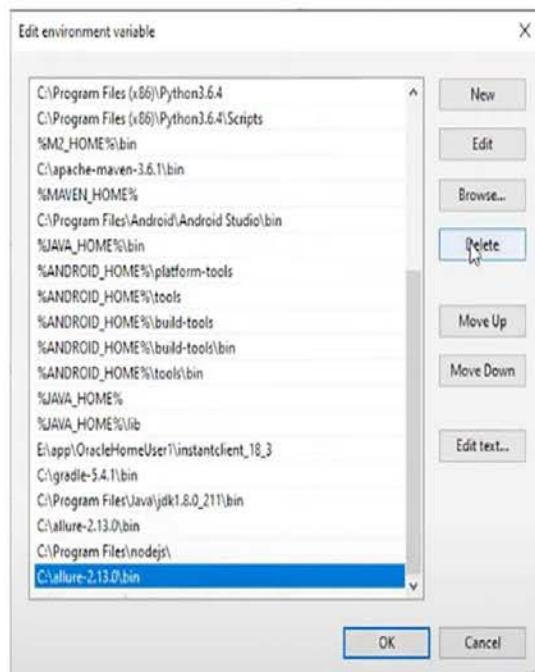


Figure 11.2: Allure Windows Installation – Adding Environment Variable

Decorators in Allure reporting

Allure provides annotations and decorators to add additional information to the test results, making them more informative and insightful. These annotations and decorators provide more context and details to the test results, making it easier to understand the test outcome.

Here are some common decorators in Allure:

- **@Step:** This annotation is used to add a description of the test step that was executed. This can be useful for creating detailed reports that show the exact steps that were taken during the test.
- **@Description:** This annotation as in the following code, is used to provide an explanation of the test. This can be useful for documenting the purpose of the test and what it is testing.
- **@Parameter:** This annotation is used to specify the parameters that were used in the test. This can be useful for debugging tests and identifying issues related to specific input values.

- **@Severity:** This decorator, as in the following code, is used to indicate the severity level of the test. Severity levels can include things such as “critical,” “high,” “medium,” and “low.” This can be useful for prioritizing tests and identifying critical issues that need to be addressed immediately.
- **@Issue:** This decorator is used to link the test to a specific issue or bug. This can be useful for tracking issues and ensuring that they are properly documented and addressed.
- **@Epic:** This decorator is used to label a test with an “epic.” Epics are used to group related tests together, making it easier to manage large test suites.
- **@Feature:** This decorator is used to label a test with a specific feature or functionality. This can be useful for identifying which features are being tested and which tests are related to specific features.

Screenshot with Allure Report

The `allure.attach()` method is used to attach the screenshot to the Allure report. The method takes three arguments: the screenshot, the name of the attachment, and the attachment type. In the following code in line 39, the name is “`subscribe_submitted`” and the `attachment_type` is PNG. When the test is executed, the screenshots are added to the reports.

Executing test and opening Allure Report

As shown in the following code, there are three tests within the class `TestAllure`, named `test_login`, `test_skiptheTest`, and `test_navigateToMenAndSubscribe`. To execute the pytest with allure reports, we need to add the flag `-alluredir <specify the directory to store reports>`. When the test is executed, it generates allure reports in the specified directory as shown in *Figure 11.3*.

Refer to the following code:

CODE – test_allure.py

1. from time import sleep
2. import pytest
3. from allure_commons.types import AttachmentType
4. from selenium import webdriver
5. from selenium.webdriver.chrome.service import Service
6. from selenium.webdriver.common.by import By
7. from webdriver_manager.chrome import ChromeDriverManager

```
8. import allure
9.
10.
11. @allure.severity(allure.severity_level.CRITICAL)
12. class TestAllure:
13.
14.     @allure.description("Test Application Valid Login")
15.     @allure.severity(allure.severity_level.CRITICAL)
16.     def test_login(self):
17.         driver = webdriver.Chrome(service=Service(ChromeDriver-
Manager("").install()))
18.         driver.get("https://magento.softwaretestingboard.com/
customer/account/login")
19.         driver.find_element(By.CSS_SELECTOR, "#email").send_
keys("admin12@gmail.com")
20.         driver.find_element(By.ID, "pass").send_keys("admin12!@")
21.         driver.find_element(By.NAME, "send").click()
22.         sleep(5)
23.
24.     @allure.severity(allure.severity_level.MINOR)
25.     def test_skipTheTest(self):
26.         pytest.skip("This test will be skipped")
27.
28.     @allure.severity(allure.severity_level.NORMAL)
29.     def test_navigateToMenAndSubscribe(self):
30.         driver = webdriver.Chrome(service=Service(ChromeDriver-
Manager("").install()))
31.         driver.get("https://magento.softwaretestingboard.com/
customer/account/login")
32.         driver.find_element(By.CSS_SELECTOR, "#email").send_
keys("admin12@gmail.com")
33.         driver.find_element(By.ID, "pass").send_keys("admin12!@")
```

```

34.     driver.find_element(By.NAME, "send").click()
35.     sleep(5)
36.     driver.find_element(By.XPATH, "//span[text()='Men']").click()
37.     driver.find_element(By.XPATH, "//input[@type='email']").send_keys("adminadmin@gmail.com")
38.     driver.find_element(By.XPATH, "//button[@title='Subscribe' and @type='submit']").click()
39.     allure.attach(driver.get_screenshot_as_png(), name="subscribe_submitted", attachment_type=AttachmentType.PNG)

```

OUTPUT

The output can be seen in *Figure 11.3*:

```
(venv) yogashivam@yogeshivam-MacBook-Pro:~$ pytest -v -c allureDir=/Users/yogashivam@yogeshivam/PycharmProjects/Frameworks/PytestDemo/reports" /Users/yogashivam@yogeshivam/PycharmProjects/Frameworks/PytestDemo/test_allure.py
=====
test session starts =====
platform darwin -- Python 3.10.7, pytest-7.2.1, pluggy-0.9.0 -- /Users/yogashivam@yogeshivam/PycharmProjects/Frameworks/venv/bin/python
cachedir: .pytest_cache
metadata: {'Python': '3.10.7', 'Platform': 'macOS-12.6-x86_64-1586-64bit', 'Packages': {'pytest': '7.2.1', 'pluggy': '1.0.4'}, 'Plugins': {'allure-pytest': '2.13.1', 'html': '3.0.2', 'netdata': '2.0.4', 'xml': '0.4.1', 'xunit': '0.1.0'}, 'Rootdir': '/Library/Java/JavaVirtualMachines/jdk-17.0.4.1.jdk/Contents/Home'}
rootdir: /Users/yogashivam@yogeshivam/PycharmProjects/Frameworks/PytestDemo, configfile: pytest.ini
plugins: allure-pytest-2.13.1, html-3.2.0, metadata-2.0.4
collected 3 items

../PytestDemo/test_allure.py::testAllure::test_login PASSED
../PytestDemo/test_allure.py::testAllure::test_skipTheTest SKIPPED (This test will be skipped)
../PytestDemo/test_allure.py::testAllure::test_navigateToHomeAndSubscribe PASSED

=====
 3 passed, 1 skipped in 23.67s =====
```

Figure 11.3: Output

Refer to the following figure:

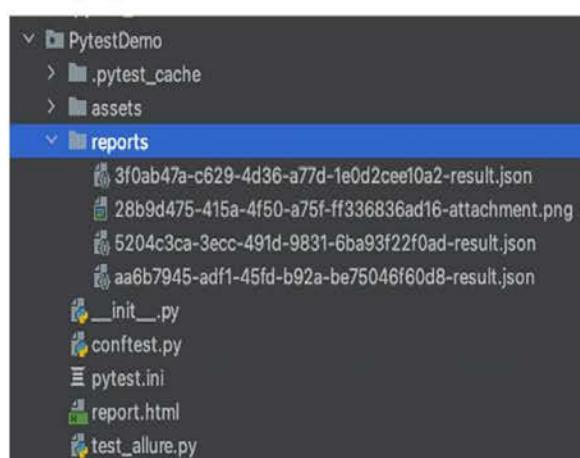


Figure 11.4: Allure reports generated.

To open the reports, navigate to the terminal and type the command `allure serve <path to reports>` such as “`allure serve /Users/yogashivamathivanan/PycharmProjects/Frameworks/PytestDemo/reports`” as shown in *Figure 11.5*:

```
yogashivamathivanan — java -classpath ~/allure-2.9.0/lib/allure-commandline-2.9.0.jar:/Users/yogashivamath...
yogashivamathivanan@Yogashivas-MacBook-Pro ~ % allure --version
2.9.0
yogashivamathivanan@Yogashivas-MacBook-Pro ~ % allure serve /Users/yogashivamathivanan/PycharmProjects/Frameworks/
/PytestDemo/reports
Generating report to temp directory...
Report successfully generated to /var/folders/lg/gb918f3j2p7cl2s4d5g26l6r0000gn/T/629112657778404419/allure-report
Starting web server...
2023-03-23 00:27:05.442:INFO::main: Logging initialized @2059ms to org.eclipse.jetty.util.log.StderrLog
Server started at <http://127.0.0.1:57463/>. Press <Ctrl+C> to exit
||
```

Figure 11.5: Open Allure reports using the command.

This will open the browser with the complete test report, annotations, and screenshots as shown in *Figure 11.6*:

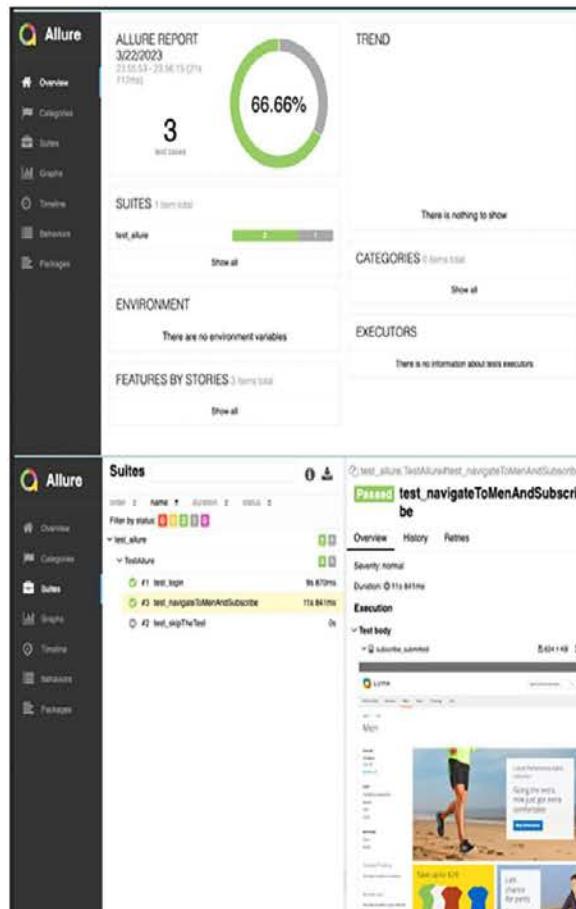




Figure 11.6: Allure reports

Designing Appium Framework

Previously, we have learned about the benefits of using Appium for mobile application automation, how it works, and support for both Android and iOS platforms. Let us start with designing the framework for mobile application automation using Appium. When designing an Appium framework, there are several key components to consider, which start with defining the project folder structure for the components of the framework as we designed for automation framework using Selenium in the previous chapters.

Folder structure

The Appium project structure is shown in *Figure 11.7*. A well-designed folder structure is critical to the success of an Appium framework. The folder structure should be organized and easy to navigate, making it easy to locate files and resources.

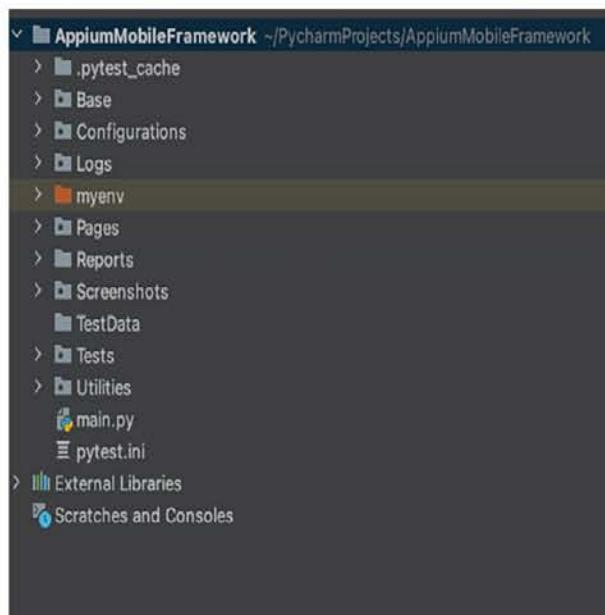


Figure 11.7: Appium Framework Folder Structures

Appium framework includes the following folders:

- **Configurations:** The configuration folder contains the configuration files that are necessary for the Appium framework such as environment variables, test configurations from the config file, authorization, and test data, and so

on. By storing these files in a separate folder, it becomes easy to modify the configuration of the framework without modifying the code.

- **Pages:** The Pages folder contains Page Objects for mobile applications. Page objects are classes that represent the UI elements of different screens and elements of the mobile application. Each page object also contains functions that interact with the elements on the screen, such as tapping on the element, swiping, scrolling, and so on. The pages folder makes the test code more organized and easier to maintain.
- **Utilities:** The utility folder contains utility classes and functions that are used across the framework, such as logging, parsing, error handling, data preparation, reading config properties, and so on. By storing these functions in a separate folder, it becomes easy to reuse them across the framework.
- **Reports:** The HTML reports from Allure, or pytest-html generated by the framework tests are stored in this folder.
- **Screenshots:** The screenshots taken during the test execution are stored in this folder. These screenshots can be used to diagnose issues and documentation.
- **Tests:** This folder contains the test scripts or test cases that will be executed by the framework. Each test case should be organized, based on its functionality and should include clear test steps.
- **Base:** This folder contains the base class for the Appium driver, and the desired capabilities used to set up the Appium driver. These methods are executed as before/setup step to set up the test environment or after/teardown step to clean up any resources after the test execution.

Implementation of Appium Framework

As we have designed the outline of the folder structure, we can start the implementation of each of the files within each folder and also implement a test and then execute the test. We will use the **Page Object Model (POM)**, which is used to represent the different screens or pages of the application under test. Each page object represents a single screen of the application with all the element locators and actions that can be performed on the element.

Driver Initialization

Let us start with the Base folder and create a Driver class that provides methods, `getAndroidDriverInstance()` and `getIOSDriverInstance()`, which returns an instance of the Appium WebDriver for Android and iOS platforms respectively. The desired capabilities are created, which define the desired behavior of the Appium server such as the platform name, version, device name, app package name, and activity. The remote method of the webdriver module is called to create a new instance

of the driver with the appium server connect URL and the desired capabilities. The created driver instance is then returned. The code for driver initialization is shown in the following `Driver.py` class.

Base Page and Screenshots

Within Base Folder, we will create another class that provides a reusable set of methods that can be used by the page objects to interact with the elements on the application page and retrieve information from it. We can use the `_init_()` method to initialize the object with the driver instance passed to it, which creates a logger object, sets an explicit wait object with a timeout of 25 seconds and a polling frequency of 1 second. The `click_element_action()` method takes the element locator, explicitly waits for the element, and then clicks on it. If the element is not located, it logs an error message. Similarly `typeText_action` types the text in the text element `elementEnabledStatus()`, and `elementDisplayedStatus()` returns Boolean, indicating the element is Enabled and Displayed respectively. The class also has methods to take screenshots using the driver “`get_screenshot()`” which attaches the screenshot to the Screenshot folder and also using the allure reporting tool “`get_allure_screenshot()`” which attaches the screenshot to the allure report. This class contains such basic methods but more such base methods can be added to it. The code for `BasePage.py` is as follows:

CODE – Driver.py

```
11.                     'appActivity': 'com.code2lead.  
                           kwad.MainActivity'}  
12.     driver = webdriver.Remote('http://localhost:4723/wd/hub',  
                               desired_capabilities)  
13.     return driver  
14.  
15.     def getIOSDriverInstance(self):  
16.         desired_capabilities = {'platformName': 'Android',  
17.                                     'platformVersion': '13',  
18.                                     'deviceName': 'Pixel 6 Pro API  
                           33',  
19.                                     'app': '/Users/  
                           yogashivamathivanan/Downloads/  
                           Android_Demo_App.apk',  
20.                                     'appPackage': 'com.code2lead.kwad',  
21.                                     'appActivity': 'com.code2lead.  
                           kwad.MainActivity'}  
22.     driver = webdriver.Remote('http://localhost:4723/wd/hub',  
                               desired_capabilities)  
23.     return driver
```

CODE – BasePage.py

```
1. import time  
2. import allure  
3. from allure_commons.types import AttachmentType  
4. from selenium.common import ElementNotVisibleException,  
   NoSuchElementException  
5. from selenium.webdriver.support.wait import WebDriverWait  
6. from selenium.webdriver.support import expected_conditions as ec  
7. import Utilities.PyLogging as alog  
8.  
9.  
10. class BasePage:  
11.
```

```
12.     def __init__(self, driver):
13.         self.driver = driver
14.         self.log = alog.appLogger()
15.         self.element = None
16.         self.explicitWait = WebDriverWait(self.driver, 25, poll_
frequency=1,
17.                                         ignored_exceptions=[Element-
NotVisibleException, No-
SuchElementException])
18.
19.     def click_element_action(self, element):
20.         try:
21.             self.explicitWait.until(ec.presence_of_element_
located(element)).click()
22.             self.log.info("Element is successfully Located and
Clicked")
23.         except:
24.             self.log.error("Element is not Located")
25.
26.     def typeText_action(self, element, typeText):
27.         try:
28.             self.explicitWait.until(ec.presence_of_element_
located(element)).send_keys(typeText)
29.             self.log.info("Element is successfully Located and
text is typed")
30.         except:
31.             self.log.error(f"Element is not Located to type the
text {typeText}")
32.
33.     def elementEnabledStatus(self, element):
34.         elementEnabledStatus = None
35.         try:
36.             elementEnabledStatus = self.explicitWait.until(ec.
```

```
37.         self.log.info("Element is in Enabled Status")
38.     except:
39.         self.log.error("Element is not Located to check the
40.                         status of element enabled or not")
41.
42.     def elementDisplayedStatus(self, element):
43.         elementDisplayedStatus = None
44.         try:
45.             elementDisplayedStatus = self.explicitWait.until(ec.
46.                                         presence_of_element_located(element)).is_displayed()
47.             self.log.info("Element is Displayed")
48.         except:
49.             self.log.error("Element is not Located to check the
50.                             status of element displayed or not")
51.
52.         return elementDisplayedStatus
53.
54.     def get_pageTitle(self):
55.         return self.driver.title
56.
57.
58.     def get_screenshot(self, screenshot_name):
59.         fileName = screenshot_name + " " + (time.
60.                                         strftime("%d_%m_%y_%H_%M_%S")) + ".png"
61.         screenshotDirectory = "./screenshots/"
62.         screenshotPath = screenshotDirectory + fileName
63.         try:
64.             self.driver.save_screenshot(screenshotPath)
65.             self.log.info("Screenshot saved to the Path : " +
66.                         screenshotPath)
67.
68.         except:
```

```
63.         self.log.error("Unable to save Screenshot to the Path  
: " + screenshotPath)
```

```
64.  
65.     def get_allure_screenshot(self, screenshotText):  
66.         allure.attach(self.driver.get_screenshot_as_png(),  
name=screenshotText,  
67.                         attachment_type=AttachmentType.PNG)
```

Configurations and Utilities

Within the configuration folder, we can add the configuration file in ".ini" format. This format is used to store the configuration data for various applications. In the following code, we have added sections "[credentials]" and "[ContactUsForm]", which are used to provide test data for the test execution. The credentials section contains a username and password, the ContactUsForm contains details for the "form" test. The configuration file stores data to be used by the Python code.

In the utility folder, we can add the property file to read these config values using configparser "**readProp.py**", as in the following code. Along with the read property file, we will have the logger utility file "**PyLogging.py**". The code has two functions, **addLogger()** and **allureLogs()**. The **appLogger()** function creates a logger object using the "logging" module, which is a standard Python module that provides a flexible way to log messages from Python code. The function uses **inspect.stack()** to get the name of the calling function, which is then used as the name of the logger. The logger object is configured to write the log messages to a file named "**ApplicationLog.log**" in the "**./Logs**" directory. After formatting, the "**appLogger()**" function returns the logger object. The **allureLogs()** function is a utility function for Allure test reporting. It creates a new step in the Allure report with the given "text" as the name of the step. The "pass" statement is included as a placeholder for the step's content, which can be added later using other Allure reporting functions.

CODE – config.ini

```
1. [credentials]  
2. username = admin@gmail.com  
3. password = admin123  
4.  
5.
```

6. [ContactUsFrom]

7. name = Shiva

```
8. email = admin@gmail.com  
9. address = 123 Main St  
10. phone = 4567890123
```

CODE – readProp.py

```
1. import configparser  
2.  
3. config = configparser.RawConfigParser()  
4. config.read("./Configurations/config.ini")  
5.  
6. username_qa = config.get('credentials', 'username')  
7. password_qa = config.get('credentials', 'password')  
8.  
9.  
10. contactName = config.get('ContactUsFrom', 'name')  
11. contactEmail = config.get('ContactUsFrom', 'email')  
12. contactAddress = config.get('ContactUsFrom', 'address')  
contactPhone = config.get('ContactUsFrom', 'phone')
```

CODE – PyLogger.py

```
1. import inspect  
2. import logging  
3. import allure  
4.  
5.  
6. def appLogger():  
7.     logName = inspect.stack()[1][3]  
8.  
9.     logger = logging.getLogger(logName)  
10.  
11.    logger.setLevel(logging.DEBUG)  
12.  
13.    file_handler = logging.FileHandler("./Logs/ApplicationLog.log")
```



```
14.  
15.     file_handler.setLevel(logging.DEBUG)  
16.  
17.     formatter = logging.Formatter('%(asctime)s - %(name)s -  
18.                                     %(levelname)s : %(message)s',  
19.                                     datefmt='%d/%m/%y %I:%M:%S %p %A')  
20.     file_handler.setFormatter(formatter)  
21.  
22.     logger.addHandler(file_handler)  
23.  
24.     return logger  
25.  
26.  
27. def allureLogs(text):  
28.     with allure.step(text):  
29.         Pass
```

Pages

The most important folder in the framework is Pages, which has the page classes that are part of a POM framework. In the following code, there are three basic pages defined “Home”, “Login” and “ContactPage” and contain all the web elements and actions. The BasePage explained previously is used as a parent class for these page classes. All three classes have an init function to initialize the constructor with the driver.

CODE – HomePage.py

```
1. from appium.webdriver.common.appiumby import AppiumBy  
2. from Base.BasePage import BasePage  
3.  
4.  
5. class Home(BasePage):
```

```
6.     loginIn_button_loc = (AppiumBy.ANDROID_UIAUTOMATOR,
    'text("LOGIN")')
7.     tabActivity = (AppiumBy.ANDROID_UIAUTOMATOR, 'text("TAB
ACTIVITY")')
8.     contactUsForm = (AppiumBy.ANDROID_UIAUTOMATOR, 'text("CONTACT
US FORM")')
9.
10.    def __init__(self, driver):
11.        super().__init__(driver)
12.
13.    def clickLogin(self):
14.        BasePage.click_element_action(self, element=self.loginIn_
button_loc)
15.
16.    def isLoginButtonDisplayed(self):
17.        return BasePage.elementDisplayedStatus(self,
element=self.loginIn_button_loc)
18.
19.    def clickTabActivity(self):
20.        BasePage.click_element_action(self, element=self.
tabActivity)
21.
22.    def clickContactUsForm(self):
23.        BasePage.click_element_action(self, element=self.
contactUsForm)
```

CODE – LoginPage.py

```
1. from appium.webdriver.common.appiumby import AppiumBy
2. from Base.BasePage import BasePage
3.
4.
5. class Login(BasePage):
6.     email_loc = (AppiumBy.ANDROID_UIAUTOMATOR, 'text("Enter
Email")')
```

```
7.     password_loc = (AppiumBy.ANDROID_UIAUTOMATOR, 'text("Enter  
9.      Password")')  
8.     loginButton = (AppiumBy.ANDROID_UIAUTOMATOR, 'text("LOGIN")')  
10.    def __init__(self, driver):  
11.        super().__init__(driver)  
12.  
13.    def enterUsernamePasswordAndLogin(self, username, password):  
14.        BasePage.typeText_action(self, element=self.email_loc,  
15.                                    typeText=username)  
16.        BasePage.typeText_action(self, element=self.password_loc,  
17.                                    typeText=password)  
18.        BasePage.click_element_action(self, element=self.  
19.                                         loginButton)
```

CODE – ContactUsPage.py

```
1. from appium.webdriver.common.appiumby import AppiumBy  
2. from Base.BasePage import BasePage  
3.  
4.  
5. class ContactPage(BasePage):  
6.     enter_Name = (AppiumBy.ANDROID_UIAUTOMATOR, 'text("Enter  
7.       Name")')  
8.     enter_Email = (AppiumBy.ANDROID_UIAUTOMATOR, 'text("Enter  
9.       Email")')  
10.    enter_Address = (AppiumBy.ANDROID_UIAUTOMATOR, 'text("Enter  
11.      Address")')  
12.    enter_MobileNumber = (AppiumBy.ANDROID_UIAUTOMATOR,  
13.                            'text("Enter Mobile No")')  
14.    submitButton = (AppiumBy.ANDROID_UIAUTOMATOR,  
15.                      'text("SUBMIT")')  
16.  
17.    def __init__(self, driver):  
18.        super().__init__(driver)
```



```
14.  
15.     def enterName(self, name):  
16.         BasePage.typeText_action(self, element=self.enter_Name,  
17.                                     typeText=name)  
18.     def enterEmail(self, email):  
19.         BasePage.typeText_action(self, element=self.enter_Email,  
20.                                     typeText=email)  
21.     def enterAddress(self, address):  
22.         BasePage.typeText_action(self, element=self.enter_  
23.                                     Address, typeText=address)  
24.     def enterNumber(self, phoneNumber):  
25.         BasePage.typeText_action(self, element=self.enter_  
26.                                     MobileNumber, typeText=phoneNumber)  
27.     def clickSubmitButton(self):  
28.         BasePage.click_element_action(self, element=self.  
                                         submitButton)
```

Tests and Execution

We have a tests folder for creating test cases where test functions can be written in the test cases using pytest syntax. We have utilities, configuration, and base pages ready to be utilized within the test scripts. We can start by importing the libraries such as Pytest, allure, created utilities, and the page objects based on defined test functions. Next, a fixture is defined to set up the test environment. In this fixture, the driver is initialized based on the desired capabilities defined in the configuration file. The fixture is then used as a parameter in the test functions. The code for the fixture is shown in the following “conftest.py”.

In the following code, there are two test files, “**test_applicationLogin**” and “**test_contactForm**”. Each test function then defines the test steps for a specific scenario. The test steps use the page objects to simulate user actions on the screen, such as clicking a button or entering text in a text field. The test function also includes

assertion statements to verify that the expected results are obtained. In addition to the test steps, the test functions also include logging statements to capture information about the test execution. The logging statements can be used to troubleshoot issues and analyze test results. After the test steps are defined, the test function includes an optional step to take a screenshot using the utility function defined in the utility file. This step can be useful for debugging and reporting test results. Finally, the test function includes a cleanup step to close the application or driver instance and release any resources. The test function also generates an allure report using the allure reporting framework.

The `test_applicationLogin.py` class has two test functions, `test_loginButton_displayed()`, which checks if the login button is displayed on the Home Page, and `test_successful_login()`, which checks for a successful login scenario. Both test functions are using the “**PyLogging**” utility function “`appLogger()`” which uses the Python inbuilt “`logging`” module to log test steps with log levels, and “`allureLogs()`” logs to allure reports. Additionally, the second test function takes the screenshot using the `get_screenshot()` method in `BasePage` class, and the test function in “`test_contactForm.py`” takes the screenshot using the “`get_allure_screenshot`” method in `BasePage`.

The tests can be written for the same application in IOS as well, when the “`@pytest.fixture(params=[])`” decorator in `pytest` is used with both ‘Android’ and ‘IOS’. When the fixture is used in the test, it will be called once for each parameter value specified in the ‘params’ argument. When a test function uses the `fixtureSetup()`, it will be called twice: once for ‘Android’ and once for ‘IOS’. The ‘request’ parameter provides information about the parameter value being used. In the `test_applicationLogin.py` test `test_successful_login()`, the commented line from 26 to 28 will be called during the run when the parameter is “IOS”. Hence, the same test can be used to execute the test in both Android and IOS.

CODE – conftest.py

```

1. import pytest
2. from Base.Driver import Driver
3.
4.
5. @pytest.fixture(params=['Android', 'IOS'], scope="class")
6. def fixtureSetup(request):
7.     driver = None

```

```
8.     if request.param == "Android":  
9.         driver = Driver().getAndroidDriverInstance()  
10.        return driver  
11.    elif request.param == "IOS":  
12.        driver = Driver().getIOSDriverInstance()  
13.        return driver
```

CODE – test_applicationLogin.py

```
1. import Utilities.PyLogging as alLog  
2. import Utilities.readProp as propRead  
3. from Pages.HomePage import Home  
4. from Pages.LoginPage import Login  
5. from Base.BasePage import BasePage  
6.  
7.  
8. class Test_AppHome:  
9.  
10.    def test_loginbutton_displayed(self, fixtureSetup):  
11.        log = alLog.appLogger()  
12.        driver = fixtureSetup  
13.        log.info("Launching Application")  
14.        assert Home(driver).isLoginButtonDisplayed(), "The Login  
           Button is Not displayed"  
15.        allog.allureLogs("The login button is displayed and  
           verified successfully")  
16.  
17.    def test_successful_login(self, fixtureSetup):  
18.        log = alLog.appLogger()  
19.        if fixtureSetup == 'Android':  
20.            driver = fixtureSetup  
21.            log.info("Launching Android Application")  
22.            Home(driver).clickLogin()
```

```
23.         Login(driver).enterUsernamePasswordAndLogin(propRead.  
username_qa, propRead.password_qa)  
24.         alLog.allureLogs("Successfully entered username and  
password, Logged in")  
25.         BasePage(driver).get_screenshot("AfterLoginPage")  
26.     # if fixtureSetup == 'IOS':  
27.         #     driver = fixtureSetup  
28.         #     log.info("Launching IOS Application")
```

CODE – test_contactForm.py

```
1. import allure  
2. from allure_commons.types import AttachmentType  
3.  
4. from Base.Driver import Driver  
5. import Utilities.PyLogging as alLog  
6. from Pages.HomePage import Home  
7. from Base.BasePage import BasePage  
8. from Pages.ContactUsPage import ContactPage  
9. import Utilities.readProp as propRead  
10.  
11.  
12. class Test_AppHome:  
13.  
14.     def test_valid_contactFormSubmission(self, fixtureSetup):  
15.         log = alLog.appLogger()  
16.         driver = fixtureSetup  
17.         log.info("Launching Application")  
18.         Home(driver).clickContactUsForm()  
19.         ContactPage(driver).enterName(propRead.contactName)  
20.         ContactPage(driver).enterEmail(propRead.contactEmail)  
21.         ContactPage(driver).enterAddress(propRead.contactAddress)  
22.         ContactPage(driver).enterNumber(propRead.contactPhone)
```

```

23.     ContactPage(driver).clickSubmitButton()
24.     BasePage(driver).get_allure_
        screenshot("ContactUsDetailsSubmitted")
25.     alLog.allureLogs("Successfully entered contact us form
        details and submitted")

```

Execution of tests and output

The tests are ready; we can execute the tests using the command “`pytest -v -s --alluredir='./Reports/Allure_Reports'` and the tests as shown in *Figure 11.8*. It will execute the tests with allure reports stored in the specified path:

```

(yeyen) yogashivamathivanan@yogashivamathivanan-MacBook-Pro:AppiumMobileFramework % pytest -v -s --alluredir='./Reports/Allure_Reports' Tests
=====
platform darwin -- Python 3.10.7, pytest-7.2.1, pluggy-1.6.0 ... /Users/yogashivamathivanan/PycharmProjects/AppiumMobileFramework/yeyen/bin/python
cachedir: .pytest_cache
rootdir: /Users/yogashivamathivanan/PycharmProjects/AppiumMobileFramework, configfile: pytest.ini
plugins: allure-pytest-2.13.1
collected 3 items

Tests/test_applicationlogin.py::Test_AppHome::test_loginbutton_displayed[Android] PASSED
Tests/test_applicationlogin.py::Test_AppHome::test_successful_login[Android] PASSED
Tests/test_contactform.py::Test_AppHome::test_valid_contactFormSubmission[Android] PASSED

3 passed in 23.87s

```

Figure 11.8: Test execution

The logging module will create the `ApplicationLog.log` file for all the logs and allure reports as shown in *Figure 11.9*:

```

(yeyen) yogashivamathivanan@yogashivamathivanan-MacBook-Pro:AppiumMobileFramework % pytest -v -s --alluredir='./Reports/Allure_Reports' Tests
=====
platform darwin -- Python 3.10.7, pytest-7.2.1, pluggy-1.6.0 ... /Users/yogashivamathivanan/PycharmProjects/AppiumMobileFramework/yeyen/bin/python
cachedir: .pytest_cache
rootdir: /Users/yogashivamathivanan/PycharmProjects/AppiumMobileFramework, configfile: pytest.ini
plugins: allure-pytest-2.13.1
collected 3 items

Tests/test_applicationlogin.py::Test_AppHome::test_loginbutton_displayed[Android] PASSED
Tests/test_applicationlogin.py::Test_AppHome::test_successful_login[Android] PASSED
Tests/test_contactform.py::Test_AppHome::test_valid_contactFormSubmission[Android] PASSED

3 passed in 23.87s

```

Figure 11.9: Test execution output logs and Allure report

Once we serve the allure report using “`allure serve /Users/yogashivamathivanan/PycharmProjects/AppiumMobileFramework/Reports/Allure_Reports`” the complete report will be opened as shown in *Figure 11.10*.

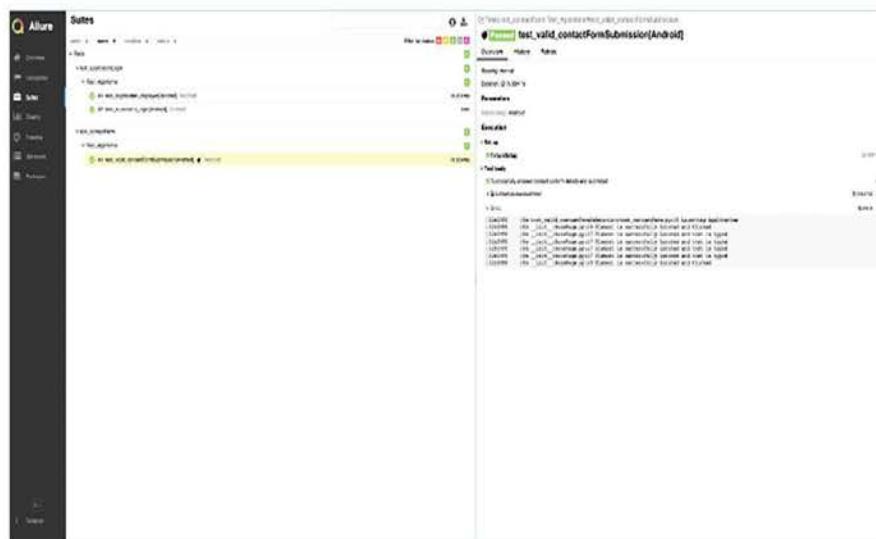


Figure 11.10: Allure report

Troubleshooting

If the test cases have warnings, as shown in *Figure 11.11*, it should be recalled that this is a known issue in the Appium community. To suppress warnings, there are multiple ways, for pytest. We can suppress warnings by using the `pytest.ini` file in the root directory with the following content. The `filterwarnings` option in the configuration file specifies a list of warning filters that should be applied during test execution. In this case, the filter is set to '`ignore::DeprecationWarning`', which tells pytest to ignore any warnings related to the "`DeprecationWarning`" class.

```
[pytest]
filterwarnings =
    # Appium team is aware of the deprecation warning - https://github.
    com/appium/python-client/issues/680
    ignore::DeprecationWarning"
```

The reason for ignoring this warning is that the Appium team is aware of the deprecation warning, and it is not relevant to the current version of the library being used. Therefore, ignoring the warning allows the test suite to run without unnecessary warnings being displayed.

Refer to the following *Figure 11.11*:



```
----- warning summary -----
Tests/test_application/login.py::test_AppHome::test_loginbutton_displayed[Android]
Tests/test_contactsForm.py::test_AppHome::test_Vt10_contactsForSubmission[Android]
  /Users/raghavreddyvrajan/PycharmProjects/login/mobile/macosx/macosx10.11/python3.6/site-packages/pytest/xdriver/webdriver.py:238: DeprecationWarning: desired_capabilities has been deprecated, please pass in an Options object with options kwargs
    super().__init__(

-- Docs: https://docs.pytest.org/en/stable/how-to/capture-warnings.html
----- 3 passed, 2 warnings in 22.82s -----
```

Figure 11.11: Deprecation Warning

Conclusion

The automation framework is of additional advantage when it comes to mobile applications. Apart from designing and developing the automation framework, we have learned about the Allure reporting tool, its installation, decorators, capturing screenshots, and how to serve the user-friendly detailed allure reports. We have discussed most of the framework utilities in the previous chapters, but in this chapter, we directed our concentration on how to design the framework considering the desired capabilities required for initializing and invoking mobile applications. We created configuration files, utility files, base pages, reports, logs, and screenshots and saw how each of these elements contributes to the overall design of the framework. This concludes our discussion on Selenium, and Appium for web and mobile automation using Python and the framework design. In the next chapter, we will look at the Selenium grid in detail, a tool in the Selenium suite that allows running tests in parallel across multiple machines and browsers at the same time.

Key facts

- Allure reporting tool provides an easy-to-use web-based interface to overview the test execution results including screenshots, logs, and error messages, which helps in quickly identifying and troubleshooting issues.
- Allure provides annotations and decorators used to add additional information to the test results, making them more informative and insightful. These annotations and decorators provide more context and details to the test results, making it easier to understand the test outcome.
- The `allure.attach()` method is used to attach the screenshot to the Allure report. The method takes three arguments: the screenshot, the name of the

attachment, and the attachment type.

- To generate the Allure report, use option “`--alluredir=<path to report folder>`” and to open the allure report use the command “`allure serve <path to reports folder>`”

- The Appium framework includes various folders such as configurations, pages, utilities, reports, screenshots, tests, and base.
- The implementation of the framework includes driver initialization, creating a BasePage and Screenshots class, configurations, and utilities.
- The Tests folder contains the actual test scripts or test cases that will be executed by the framework. Each test case should be organized based on its functionality and include clear test steps. The test scripts should use the page objects.

Questions

1. How to generate the Allure reports for the test execution? How to open the allure report using the command line?
2. What are decorators in Allure reporting? What is @Severity?
3. How to take screenshots in Allure reporting?
4. How driver is initialized in BasePage?
5. What is a `pytest.ini` file?

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



CHAPTER 12

Dockerized

Selenium Grid

Introduction

Docker! What is docker? What are containers in docker? Why is docker so famous in the software industry? How can docker be applied in test automation with Selenium? These are some of the major questions we will discuss in detail in this chapter. Docker has emerged as one of the most popular tools in software development in recent years. It is a containerization platform that allows developers to package applications, along with their dependencies, into containers that can be deployed on any system. This makes it easier to build, ship, and run applications across different environments and helps solve the age-old problem of "it works in my machine." Containers in Docker are lightweight, portable units of software that encapsulate everything an application needs, in order to run code, libraries, system

tools, and settings. They provide a consistent and isolated environment for running applications.

Selenium grid is a tool that allows running tests on multiple machines or browsers at the same time. It is useful for testing applications on different platforms, operating systems, and browser configurations in order to ensure that the application works as expected for all users. Here, Docker can be a valuable tool for managing test environments and improving the efficiency of test execution. By using Docker to set up a Selenium Grid environment, test automation engineers can run Selenium

tests in parallel on multiple nodes, which can significantly reduce test execution time. Docker also makes it easier to manage dependencies and ensure consistency between test environments, which can further help to prevent issues caused by environmental differences. We will cover the basic of docker and the steps for setting up a scalable and easily manageable Dockerized selenium grid environment using docker-compose.

Structure

In this chapter, we will discuss the following topics:

- Virtualization
 - Limitations of Virtualization Compared to Docker
- Docker
- Docker Installation
- Docker Commands
- Running Selenium Tests with Docker
- Selenium Grid
- Selenium Grid with Docker
 - Advantages of Selenium Grid
 - Running Selenium Grid in Docker

Objectives

This chapter will bring in the concept of Docker and how Docker can be used in test automation with Selenium. To understand docker and containerization, we will first understand and look at the concept of virtualization. We will look at the complete docker essentials from the introduction, installations, overview of the docker

image and Docker Hub; containerization and container management, and Docker commands, followed by Selenium grid integration with Docker. We will gather a comprehensive understanding of Docker to understand its benefits in software development and test automation with Selenium. Within Docker, we will learn about the most powerful Docker feature, that is, Docker compose, which is a tool for defining and running multi-container Docker applications. We will learn the usage of Docker compose and how we can start and stop the entire application stack with a single command, and Docker compose will automatically manage the creation and destruction of the containers.

Virtualization

Virtualization is a technology that enables running multiple operating systems or applications on the same physical machine, by simulating the hardware of a computer. This is done by using a software layer called a hypervisor that allows multiple virtual machines to run on a single physical machine. Each **Virtual Machine (VM)** operates as if it were a separate physical machine, with its own operating system, CPU, memory, and disk space. Virtualization is used in software development and test processes because it allows developers and testers to create and manage multiple environments on a single physical machine. This helps to isolate applications, prevent conflicts between software components, and facilitate testing.

For example, suppose we are developing a web application that needs to be tested on multiple operating systems and web browsers. We can create multiple virtual machines on a single physical machine, each with a different operating system and web browser combination. This allows us to test the application in various environments without needing to set up a separate physical machine for each combination.

Limitations of virtualization compared to containerization

Virtualization has several limitations that Docker helps to overcome. Some of these limitations of virtualization are:

- **Resource utilization:** Virtualization technology requires a full operating system for each virtual machine, which can result in poor resource utilization. Docker, on the other hand, allows multiple containers to share a single operating system kernel, resulting in better resource utilization and cost savings.

- **Speed:** Virtual machines have to initiate the entire boot process, including the operating system, thus making it a lengthier process. On the other hand, containers can significantly decrease the deployment and execution time of an application by quickly starting up since the operating system is already running. It saves a considerable amount of time during the application testing process.

-
- **Portability and application sharing:** Virtual machines can be heavy and require having a copy of the entire operating system, including the kernel, configuration files, all directories, system libraries, and all the utilities, thus making it difficult to share and move them between different environments. Container images are lightweight and portable, making it easy to share and move containers between development, testing, and production environments.
 - **Overhead:** Virtual machines run separate operating systems and require a significant amount of overhead in terms of memory, storage, and CPU usage. Containers consume fewer resources, resulting in better performance and scalability.
 - **Operating system requirements:** When a business needs to operate multiple applications that have fully functional dedicated operating systems, virtual machines are more suitable. However, if multiple applications require the same operating system requirement, then containers present a more viable option.
 - **Maintenance:** Managing and maintaining virtual machines can be challenging, especially when dealing with many virtual machines. Docker simplifies container management, making it easy to deploy and maintain containers at scale.

Overall, virtualization and Docker are both technologies that enable running multiple environments on a single physical machine. Virtualization simulates hardware to create isolated environments, while Docker uses containerization to package applications and dependencies into a portable container. Unlike virtual machines, Docker containers share the operating system kernel with the host machine, and do not require a hypervisor to simulate hardware. Docker provides a more efficient,

portable, and scalable way to build and deploy software applications. By reducing resource utilization, improving portability, and simplifying container management, Docker allows developers to focus on building and testing the applications, rather than worrying about the underlying infrastructure. The architectural difference is shown in the following *Figure 12.1*:

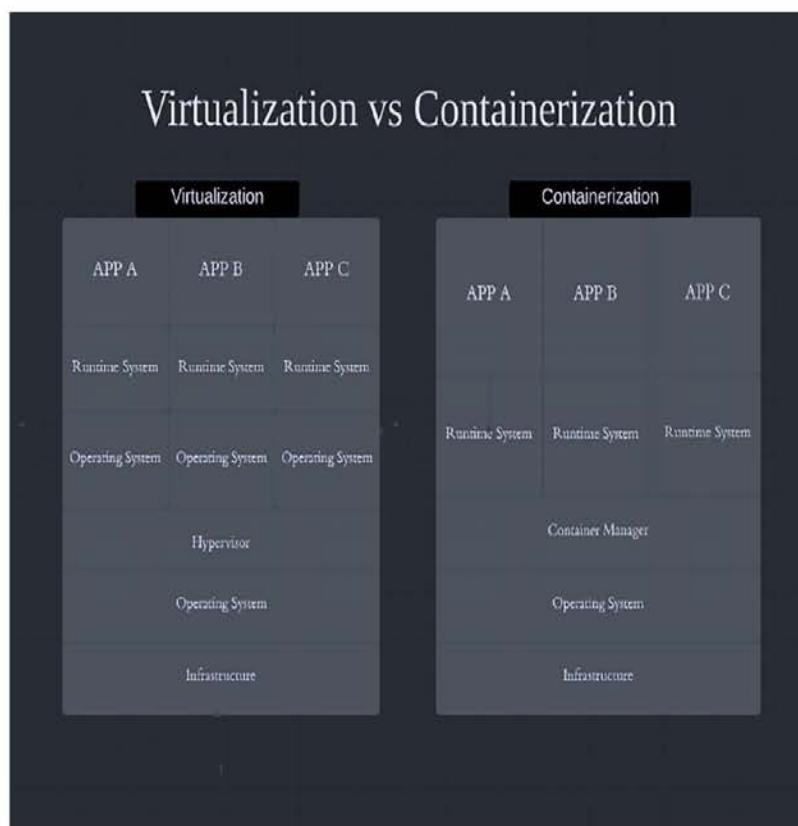


Figure 12.1: Virtualization vs Containerization

Docker

Docker is an open-source platform for developing, deploying, and running applications in containers. A container is a lightweight and standalone executable package that contains everything required to run an application. Containers provide

~~Package that contains every thing required to run an application. Containers provide~~
a consistent and reproducible environment that can be easily shared across different systems, making it easier to develop, test, and deploy software applications.

Let us look at the concept in simpler terms. Imagine you have a dish recipe. The recipe requires a specific type of tools and ingredients. You can make the dish at home, where you have all the necessary tools and ingredients. Now you want to share the dish recipe with a friend, but they might not have the same specific ingredients required, and even if they do, they might not have the required tools such as an oven. This is where Docker comes in; it allows you to package the recipe and all the necessary ingredients and tools in a container, which can be shared with your friend, who can make the dish exactly like you, without worrying about compatibility issues or missing ingredients. In software development, let us say a web application that requires a specific version of Python and a database, you can

create a docker container that includes Python, the database, and the application code and share it with others, who can run the application without worrying about installing Python or the database separately.

Advantages of Docker

In recent years, Docker has become increasingly popular in the software development industry due to its numerous benefits, including:

- **Portability:** In the agile software development industry, it is essential to be able to move the applications between different environments quickly and easily. Docker containers are self-contained and can be easily moved between different systems, making testing and deploying applications across different environments easier.
- **Isolation:** Software applications require specific runtime environments and dependencies to function properly. With traditional software development practices, it can be challenging to ensure that an application runs consistently on different machines with different configurations. Docker helps solve this problem by providing a high degree of isolation between different applications and their dependencies, reducing the likelihood of conflicts or compatibility issues.
- **Efficiency:** Docker containers are lightweight and consume fewer resources than traditional virtual machines, making it easier to run multiple containers on a single host, which results in better utilization of hardware resources and cost saving.
- **Consistency:** Docker containers provide a consistent and reproducible environment that can be easily shared between different systems, ensuring that applications work the same way in all environments.
- **Scalability:** Docker containers can be easily scaled up or down based on demand, making it easier to handle fluctuations in traffic or workload, which is essential for modern cloud-based applications.

These benefits make Docker an ideal platform for software development, especially for applications that need to be deployed across multiple systems and environments.

Docker use cases in software development

Some of the popular use cases of Docker in software development include:

- **Containerized development environments:** Docker can be used to create consistent and reproducible development environments that can be easily

shared between different developers. This makes it easier to onboard new developers and ensures that everyone is working in the same environment.

- **Continuous integration and continuous deployment (CI/CD):** Docker can be integrated with CI/CD pipelines to automate the build, test, and deployment process of software applications. This makes it easier to deploy applications faster and with greater reliability.
- **Microservices:** Docker can be used to create and deploy microservices, which are small and independent components of a larger application. This makes it easier to build and maintain complex applications that can be easily scaled and updated.
- **Cloud computing:** Docker can be used to deploy and manage applications in cloud environments, such as **Amazon Web Services (AWS)** or Microsoft Azure. This makes it easier to manage and scale applications in the cloud.

Docker terminology

Let us now go over some Docker terminology:

- **Docker Image:** The docker image is a file or executable package comprised of everything needed to run an application, including code, libraries, dependencies, and configuration files.
- **Docker Containers:** Docker containers, on the other hand, are the actual runtime instance of docker images. A docker container is a standardized unit that can be created on the fly to deploy a particular application or environment. It could be an Ubuntu container, a CentOS container, and so on. They are isolated environments that can run an application and its dependencies without interfering with the host system or other containers running on the same system. Containers are designed to be created, used, and destroyed as needed.
- **Docker Hub:** Docker Hub is a cloud-based repository service where Docker images can be stored, managed, and shared with others. Docker images can be pulled from Docker Hub and used to create Docker containers. When a Docker image is uploaded to Docker Hub, it can be shared with other users who can then pull the image and create their containers, making it easier to collaborate on projects. It also provides a central location for developers to find Docker images that can be used in their projects, reducing the time and effort required to create new images from scratch. Docker Hub also provides features such as automated builds and version control. Automated builds allow users to create Docker images automatically from source code repositories such as GitHub, while version control allows users to manage

different versions of their Docker images. The Docker hub can be accessed using the link <https://hub.docker.com/>.

The relationship flow diagram is shown in the following *Figure 12.2*. The images can be pushed or pulled from the Docker Hub registry by the Docker engine. Once a desired image is pulled, a Docker container can be created using that image, which is essentially an instance of the image running in an isolated environment with its own file system, networking, and resources.

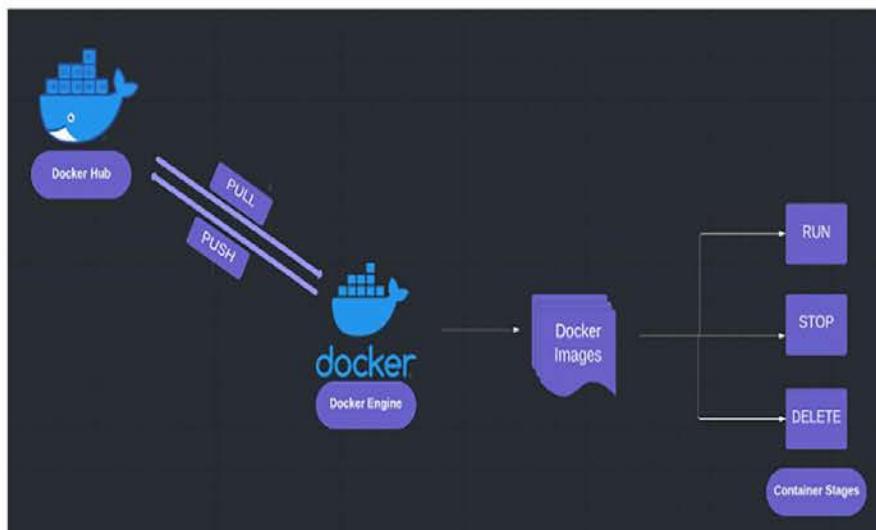


Figure 12.2: Docker Image from Docker Hub and Create the container Flow

- **Dockerfile:** Dockerfile is a file without any extension that contains instructions for building a Docker image. Docker uses these instructions to create a containerized environment that is consistent across different machines. Dockerfile includes instructions for things such as:
 - Setting a base image for the environment.
 - Adding files or directories to the environment.
 - Running commands to install and configure the software.
 - Specifying the ports that should be exposed by the container.
 - Setting environment variables.
 - Defining a default command to run when the container starts.

In the following example, the Dockerfile starts with the FROM instruction, which specifies the base image to use for the environment. In this case, we are using the official Python 3.9 runtime for Alpine Linux. Next, we use the WORKDIR instruction to set the working directory for the container to `/app`. This is where we will copy our application code. Then, we use the COPY instruction to copy the contents of the current directory into the container at the `/app` directory. Finally, we use the CMD instruction to specify the command to run when the container starts.

application code and a `requirements.txt` file) into the container's `/app` directory. After that, we use the RUN instruction to install any required packages specified in the `requirements.txt` file using pip. Next, we use the EXPOSE instruction to expose port 80 to the outside world. Then, we use the ENV instruction to set an environment variable called NAME to World. Finally, we use the CMD instruction to specify that the container should run the `app.py` file using Python when it launches.

- **Build:** Build refers to the process of creating a Docker image from a Dockerfile. When we build a Docker image, Docker reads the instructions in the Dockerfile and creates a new image. The resulting image contains all the dependencies and configurations necessary to run a particular application or service. When we run the `docker build` command with the following Dockerfile, Docker will create a new image based on the instructions in the file. This image will include Python 3.9 and any required packages, as well as our application code and the `app.py` file. We can then use this image to create a container that runs our application, which will be accessible on port 80.

CODE – Dockerfile

```
1. # Use an official Python runtime as a parent image
2. FROM python:3.9-alpine
3.
4. # Set the working directory to /app
5. WORKDIR /app
6.
7. # Copy the current directory contents into the container at /app
8. COPY . /app
9.
10. # Install any needed packages specified in requirements.txt
11. RUN pip install --no-cache-dir -r requirements.txt
12.
13. # Make port 80 available to the world outside this container
14. EXPOSE 80
15.
16. # Define environment variable
17. ENV NAME World
```

18.

19. # Run app.py when the container launches

20. CMD ["python", "app.py"]

Docker installation

Let us now install Docker.

Docker Mac installation

Navigate to Docker docs at <https://docs.docker.com/desktop/install/mac-install/> and download the Docker desktop depending on the Mac configuration. Once downloaded, open the Docker installer, then drag and drop the Docker into the Applications folder. Open Docker from the applications folder. The Docker application should start, as shown in *Figure 12.3*.

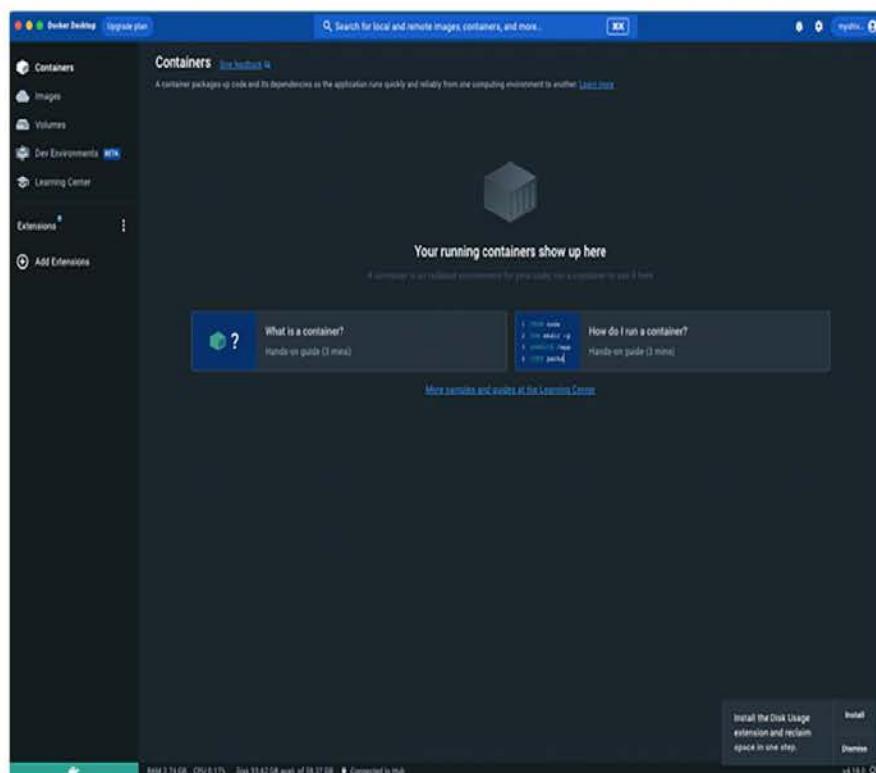


Figure 12.3: Docker Application

In the terminal, type “`docker`” and “`docker version`” and the response can be seen in *Figure 12.4*. Sign in or sign up to Docker. If there is any error in the terminal’s docker response, then restart the docker application, and it should work.

```
yogashivamathivanan@Yogashivas-MacBook-Pro ~ % docker version
Client:
  Cloud integration: v1.0.31
  Version:          20.10.24
  API version:      1.41
  Go version:       go1.19.7
  Git commit:       297e128
  Built:            Tue Apr  4 18:21:21 2023
  OS/Arch:          darwin/amd64
  Context:          default
  Experimental:    true

Server: Docker Desktop 4.18.0 (104112)
Engine:
  Version:          20.10.24
  API version:      1.41 (minimum version 1.12)
  Go version:       go1.19.7
  Git commit:       5d6db84
  Built:            Tue Apr  4 18:18:42 2023
  OS/Arch:          linux/amd64
  Experimental:    false
  containerd:
    Version:         1.6.18
    GitCommit:       2456e983eb9e37e47538f59ea18f2043c9a73640
  runc:
    Version:         1.1.4
    GitCommit:       v1.1.4-0-g5fd4c4d
  docker-init:
    Version:         0.19.0
    GitCommit:       de40ad0
```

Figure 12.4: Docker Verification

Docker Windows Installation

Navigate to Docker docs at <https://docs.docker.com/desktop/install/windows-install/> and download the Docker desktop. Open the Docker installer to install, follow the steps, and then after installation, restart the computer. After restart, the docker desktop application will open and there will be an error related to WSL 2 installation as shown in *Figure 12.5*, which means that the Windows Subsystem for Linux, WSL 2 component is not installed or is not fully installed on the machine. Click on the link prompted, as shown in *Figure 12.5* or <https://learn.microsoft.com/en-us/windows/wsl/install-manual#step-4---download-the-linux-kernel-update-package> and navigate to the page as in *Figure 12.6*. Please refer to the following figure:

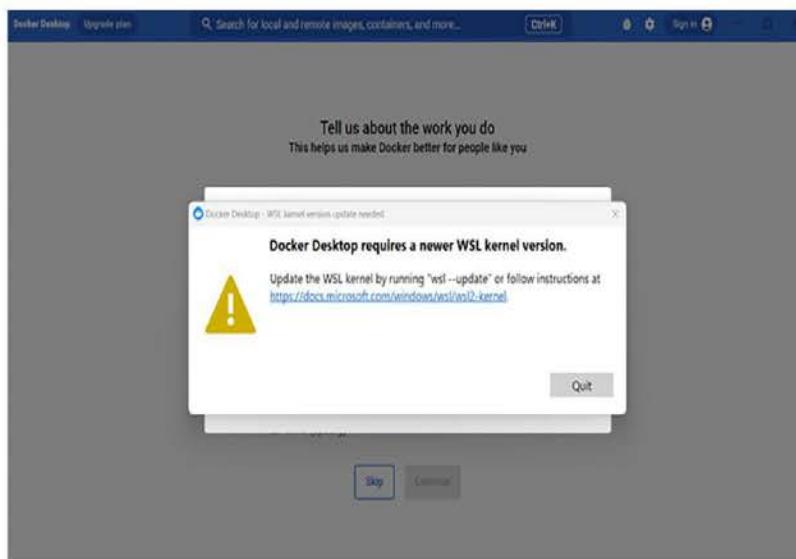


Figure 12.5: WSL2 error in docker desktop in windows

Refer to the following *Figure 12.6*:

Step 4 - Download the Linux kernel update package

1. Download the latest package:

- WSL2 Linux kernel update package for x64 machines [♂](#)

① Note

If you're using an ARM64 machine, please download the ARM64 package [♂](#) instead. If you're not sure what kind of machine you have, open Command Prompt or PowerShell and enter: `systeminfo | find "System Type"`. **Caveat:** On non-English Windows versions, you might have to modify the search text, translating the "System Type" string. You may also need to escape the quotations for the find command. For example, in German `systeminfo | find '"Systemtyp"'`.

2. Run the update package downloaded in the previous step. (Double-click to run - you will be prompted for elevated permissions, select 'yes' to approve this installation.)

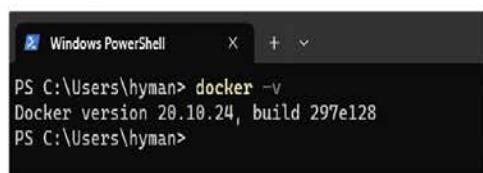
Once the installation is complete, move on to the next step - setting WSL 2 as your default version when installing new Linux distributions. (Skip this step if you want your new Linux installs to be set to WSL 1).

① Note

For more information, read the article [changes to updating the WSL2 Linux kernel](#), available on the [Windows Command Line Blog](#).

Figure 12.6: WSL2 latest package installation page.

Download the latest package “**WSL2 Linux kernel update package for x64 machines**” and restart the machines. Then run the command “**docker -v**” in the terminal, which should respond with the docker version, as shown in *Figure 12.7*:



```
Windows PowerShell
PS C:\Users\hyman> docker -v
Docker version 20.10.24, build 297e128
PS C:\Users\hyman>
```

Figure 12.7: docker -v to validate docker installation

Docker commands

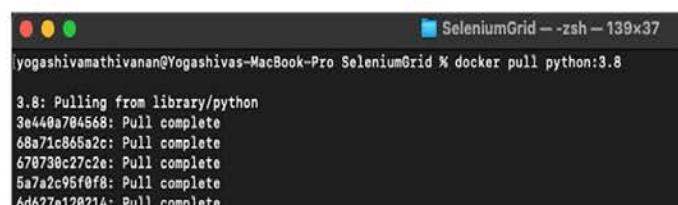
Docker commands are used to interact with the Docker engine, which is the most important component of Docker that manages containers, images, networks, and volumes. These commands are typically used in a command line interface or terminal to create, run, and manage docker containers and images. Let us explore some of the basic and useful commands using an example, so that we can understand how these commands can be useful in real-world scenarios. Here is a basic Python script with the file name “**TestPython.py**” with only a print statement “`print("Hello, World!")`” so that the example is easy to understand.

docker pull <image_name>

This command downloads the specified Docker image from a Docker registry (such as Docker Hub) and stores it on your local system.

There is an image for official Python in dockerhub at https://hub.docker.com/_/python

As shown in *Figure 12.8*, we can pull the **python3.8** image using “**docker pull python:3.8**”:



```
SeleniumGrid -- zsh -- 139x37
yogashivamivanan@Yogashivas-MacBook-Pro SeleniumGrid % docker pull python:3.8
3.8: Pulling from library/python
3e440e704568: Pull complete
68a71c865a2c: Pull complete
670730c27c2e: Pull complete
5a7a2c95f0f8: Pull complete
6d627e120214: Pull complete
```

```
f8c6dc678081: Pull complete
b64f731a273: Pull complete
e851df9ffab61: Pull complete
aa941cbc6a4c: Pull complete
Digest: sha256:f9af3c8c4614d955495d23a66576babe3b37d4310bf9c33ea1ebf9f18991d8fe
Status: Downloaded newer image for python:3.8
docker.io/library/python:3.8
yogashivamathivanan@Yogashivas-MacBook-Pro SeleniumGrid % ||
```

Figure 12.8: docker pull command

docker build -t <image_name>

This command builds a Docker image from a Dockerfile located in the current directory and tags it with the specified name (<image_name> in this case).

The Dockerfile is required to be created in the current directory along with the Python script “**TestPython.py**”. Let us break down the individual commands in the Dockerfile. The Dockerfile specifies that we want to use the Python 3.8 image as the base image, copies the contents of the current directory into a directory called **/app** in the container, and then sets the working directory to **/app**. Finally, it runs the command **python Testpython.py**. As shown in *Figure 12.9*, we can build the docker image from Dockerfile.

CODE – Dockerfile

1. FROM python:3.8
2. COPY . /app
3. WORKDIR /app
4. CMD ["python", "TestPython.py"]

```
yogashivamathivanan@Yogashivas-MacBook-Pro SeleniumGrid % docker build -t new-docker-image .
[+] Building 3.0s (8/8) FINISHED
--> [internal] Load build definition from Dockerfile
--> => transferring dockerfile: 117B
--> [internal] load .dockerignore
--> => transferring context: 2B
--> [internal] load metadata for docker.io/library/python:3.8
--> [1/3] FROM docker.io/library/python:3.8
--> [internal] load build context
--> => transferring context: 182.90kB
--> [2/3] COPY . /app
--> [3/3] WORKDIR /app
--> => expecting to image
--> => exporting layers
--> => writing image sha256:c478cc42100a996a04372f30cd1bbbff4ef7e4a9e9ada53000498d6c0711fc9
--> => making to docker.io/library/new-docker-image
yogashivamathivanan@Yogashivas-MacBook-Pro SeleniumGrid % ||
```

Figure 12.9: docker build command

We can also see the new image listed in docker desktop application in *Figure 12.10*:



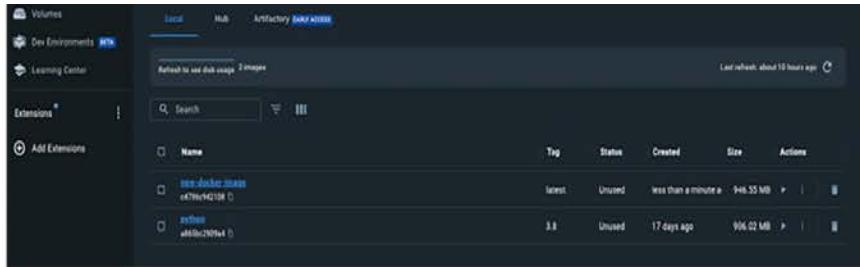


Figure 12.10: docker application image created

docker run <image_name>

This command runs a Docker container from the specified image (<image_name>). As in the Dockerfile, we have CMD to run the Python script in **TestPython.py**. When the container starts, the script will be executed when the container is run using the **docker run** command. Since the **TestPython.py** file only has the print statement, the output of the run command is “Hello, World!” as shown in *Figure 12.11*:

```
SeleniumGrid -- zsh - 139x22
[yogashivamathivanan@Yogashivas-MacBook-Pro SeleniumGrid % docker run new-docker-image
Hello, World!
yogashivamathivanan@Yogashivas-MacBook-Pro SeleniumGrid % ]
```

Figure 12.11: docker run command

docker ps

This command lists all running Docker containers. As shown in *Figure 12.12*, the flag “-a” is used to list all the containers including the ones that are stopped. This command shows the Container ID, the image used to create the container, the command that was run in the container, the container’s status, the time when the container was created, and the container’s name.

```
SeleniumGrid -- zsh - 139x22
[yogashivamathivanan@Yogashivas-MacBook-Pro SeleniumGrid % docker ps -a
CONTAINER ID   IMAGE      COMMAND   CREATED     STATUS      PORTS     NAMES
1016750d42a0   new-docker-image   "python TestPython.py"   5 minutes ago   Exited (0) 5 minutes ago          jovial_wright
yogashivamathivanan@Yogashivas-MacBook-Pro SeleniumGrid % ]
```

Figure 12.12: docker ps -a command

docker stop <container_id>

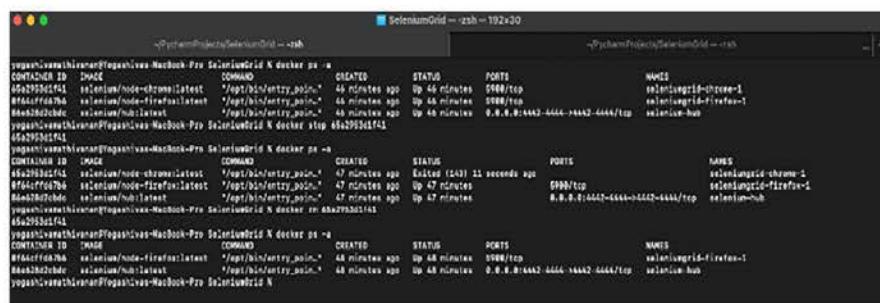
The command stops a Docker container by specifying its ID or name.

This command stops a running Docker container, where <container_id> is the ID of the container (you can get this from the docker ps command). The docker stop is shown in *Figure 12.13*.

344 ■ Selenium and Appium with Python

docker rm <container_id>

This command removes a stopped Docker container, where <container_id> is the ID of the container. The docker rm is shown in *Figure 12.13*:



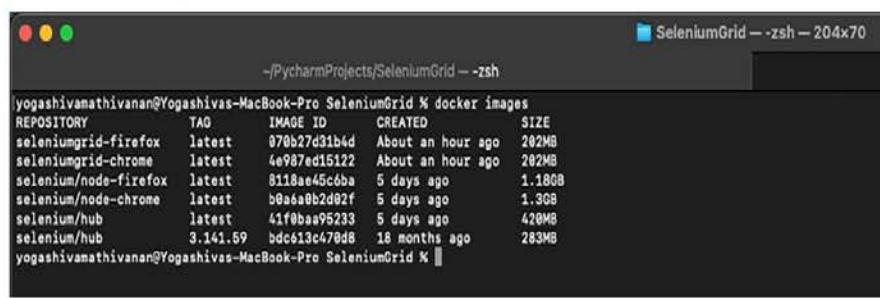
The screenshot shows two terminal windows side-by-side. Both windows have the title 'SeleniumGrid -- zsh - 192x30'. The left window displays the output of the 'docker ps' command, listing several containers with their IDs, names, and ports. The right window shows the execution of the 'docker stop' and 'docker rm' commands on a specific container.

```
yogashivamathivanan@Yogashivas-MacBook-Pro SeleniumGrid % docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS               NAMES
8f64cc7fca3a        selenium/node-chrome:latest   "/opt/bin/entry point"   44 minutes ago   Up 44 minutes          0.0.0.0:4442->4442/tcp   selenium-grid-chrome-1
8f64cc7fca3a        selenium/node-firefox:latest  "/opt/bin/entry point"   44 minutes ago   Up 44 minutes          0.0.0.0:4443->4443/tcp   selenium-grid-firefox-1
8ae38c6bde        selenium/hub:latest      "/opt/bin/entry point"   44 minutes ago   Up 44 minutes          0.0.0.0:4442-4443-4442-4444/tcp   selenium-hub
yogashivamathivanan@Yogashivas-MacBook-Pro SeleniumGrid % docker stop 65a2958d1f41
65a2958d1f41
yogashivamathivanan@Yogashivas-MacBook-Pro SeleniumGrid % docker rm 65a2958d1f41
65a2958d1f41
yogashivamathivanan@Yogashivas-MacBook-Pro SeleniumGrid % docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS               NAMES
8f64cc7fca3a        selenium/node-chrome:latest   "/opt/bin/entry point"   47 minutes ago   Exited (143) 11 seconds ago          selenium-grid-chrome-1
8f64cc7fca3a        selenium/node-firefox:latest  "/opt/bin/entry point"   47 minutes ago   Up 47 minutes          0.0.0.0:4443->4443/tcp   selenium-grid-firefox-1
8ae38c6bde        selenium/hub:latest      "/opt/bin/entry point"   47 minutes ago   Up 47 minutes          0.0.0.0:4442-4443-4442-4444/tcp   selenium-hub
yogashivamathivanan@Yogashivas-MacBook-Pro SeleniumGrid % docker rm 65a2958d1f41
65a2958d1f41
yogashivamathivanan@Yogashivas-MacBook-Pro SeleniumGrid % docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS               NAMES
8f64cc7fca3a        selenium/node-chrome:latest   "/opt/bin/entry point"   48 minutes ago   Up 48 minutes          0.0.0.0:4442->4442/tcp   selenium-grid-chrome-1
8ae38c6bde        selenium/hub:latest      "/opt/bin/entry point"   48 minutes ago   Up 48 minutes          0.0.0.0:4442-4443-4442-4444/tcp   selenium-hub
```

Figure 12.13: docker stop and rm command

docker images

This command lists all Docker images that are currently available on your system. The docker images command is shown in *Figure 12.14*:



The screenshot shows a single terminal window with the title 'SeleniumGrid -- zsh - 204x70'. It displays the output of the 'docker images' command, which lists various Docker images along with their repository, tag, image ID, creation time, and size.

```
yogashivamathivanan@Yogashivas-MacBook-Pro SeleniumGrid % docker images
REPOSITORY          TAG      IMAGE ID      CREATED             SIZE
seleniumgrid-firefox    latest   070b27d31b4d  About an hour ago  202MB
seleniumgrid-chrome    latest   4e987ed15122  About an hour ago  202MB
selenium/node-firefox   latest   8118ae45c6ba  5 days ago       1.18GB
selenium/node-chrome    latest   b0a6aab2d02f  5 days ago       1.3GB
selenium/hub           latest   41f0bba95233  5 days ago       420MB
selenium/hub           3.141.59  bdc613c470d8  18 months ago    283MB
yogashivamathivanan@Yogashivas-MacBook-Pro SeleniumGrid %
```

Figure 12.14: docker images command

docker push <image_name>

This command uploads a Docker image to a Docker registry so that it can be used by other users or machines.

Docker volume

In Docker, a volume is a persistent data storage mechanism that allows a container to store and access data outside of its filesystem. A volume is a separate entity from a container, and it can exist independently of the container that creates it. Volumes are used to share data between containers or to store data that needs to persist beyond

the life of a container. Volumes can also be used to store data that needs to be shared between multiple containers, such as log files or configuration files.

Volumes can be created and managed using Docker commands, such as `docker volume create`, `docker volume ls`, `docker volume rm`, and `docker volume inspect`. Volumes can be mounted in a container using the `--mount` or `-v` options when running a container, and the data stored in the volume can be accessed by the container just like it would access data on its own filesystem.

Docker Compose

Docker Compose is a tool that allows defining and running multi-container applications. With Compose, we can use a YAML file to configure all the application services, networks, and volumes. Then with a single `docker compose` command, we can create and start all the services from the configuration.

Let us look at an example to understand how powerful docker compose is. Let us assume we have a Python web application that uses Flask, which is a lightweight Python web framework used to create web applications. Create a project directory and create a Dockerfile in the same directory which sets the base image as `python:3.9-slim-buster`, creates a working directory `/app` in the container, copies `requirement.txt` to the working directory, installs the dependency using `pip` command, copies the rest of the application code and runs the command `python app.py` when the container is started.

CODE – Dockerfile

1. FROM python:3.9-slim-buster
2. WORKDIR /app
3. COPY requirements.txt .
4. RUN pip install --no-cache-dir -r requirements.txt

5. COPY . .
6. CMD ["python", "app.py"]

Create a file `docker-compose.yaml` in the project directory with the following contents. The file defines a service named 'web' which builds the image using the Dockerfile in the current directory ('.'), and maps port '5000' on the host to port '5000' in the container.

CODE – docker-compose.yaml

```
1. version: "3.8"  
2.  
3. services:  
4.   web:  
5.     build: .  
6.     ports:  
7.       - "5000:5000"
```

Create a file `app.py` in the project directory with the following contents. Install flask dependency using `pip install flask`, and the `requirement.txt` file is required to have flask dependency to be able to run the application in the container.

CODE – app.py

```
1. from flask import Flask  
2.  
3. app = Flask(__name__)  
4.  
5. @app.route("/")  
6. def hello():  
7.     return "Hello, Docker Compose!"  
8.  
9. if __name__ == "__main__":  
10.    app.run(host="0.0.0.0", debug=True)
```

Open the terminal and run the command `docker compose up`, as shown in *Figure 12.15*. This command starts the containers defined in the `docker-compose.yaml` file. Open the web browser and navigate to `http://localhost:5000`, and we should see the “Hello, Docker compose!” message.

Refer to *Figure 12.15*:

```
[+] Building 21.3x (18/11)
  => [internal] load build definition from Dockerfile
  => [internal] load .dockerignore
  => [internal] load .envrc
  => [internal] load context: 28
  => [internal] load metadata for docker.io/library/python:3.9-slim-buster
  => [auth] library/python:3.9-slim-buster token: 3e0236:bc1e889f98d31764033:bd197e3d959fb4cabcff86d0035fa53a3527ed 0.00
[+] 1/51 FROM docker.io/library/python:3.9-slim-buster@sha256:bc1e889f98d31764033:bd197e3d959fb4cabcff86d0035fa53a3527ed 19.4s
--> sha256:94231aa89979d011760873d43973395d99a4a67bf78840035fa53a3527ed 0.00
--> sha256:d411e42e05937715d69a6030caefb932047ff76a7e2274f49b9a0f 1.37s / 1.37s
--> sha256:9511ea2979d0c771260a0502aa29b72320cff7067e20274f49b9a0f 1.38s / 1.38s
--> sha256:2049568e21943077220a0502aa29b72320cff7067e20274f49b9a0f 27.14s / 27.14s
--> sha256:a7a0f344921b0a26c0830547e323508cfec94a3c295994086949b9a0f 2.17s / 2.17s
--> sha256:770918563eb2000560111e317403f9ec252b1377979e7a 11.18s / 11.18s
--> sha256:9423000cc97e0e99f8000554e0e999e7979e7512705996 24.5s / 24.5s
--> sha256:9445357707344405f536375336676439474e6e4a016535e0f5983 3.19s / 3.19s
--> extracting sha256:349b0e84326e337712ca62a520a59a8c4fb9d3d4a84f93d 1.4s
--> extracting sha256:c75cd07444499110504584449f4235d9aff94e4a22595d644799af 0.2s
--> extracting sha256:e1a77209403655e70487a5023a77121c2746e350cc0021623770c9374 0.1s
--> extracting sha256:4227200dc055e70487a5023a77121c2746e350cc0021623770c9374 0.1s
--> extracting sha256:914094c72594e4c91c1bf75366363d6971a8ee88616591dfc983 0.1s
--> [internal] load build context: 318.878
--> [internal] copy requirements.txt
--> [internal] COPY requirements.txt
--> [internal] RUN pip install --no-cache-dir -r requirements.txt
--> [internal] COPY
--> [internal] exporting image
--> [internal] killing image sha256:092514992616622883786c632897778544a56117820d7f1036194949581240
--> [internal] using docker.io/library/python:3.9-slim
[+] Running 2/1
  > Network seleniumgrid_default Created 0.1s
  > Container seleniumgrid-web-1 Created 0.1s
Attaching to seleniumgrid-web-1
seleniumgrid-web-1  * Receiving Flask app 'app'
seleniumgrid-web-1  * Debug mode on
seleniumgrid-web-1  * WARNING! This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
seleniumgrid-web-1  * Running on all addresses (0.0.0.0)
seleniumgrid-web-1  * Running on http://127.0.0.1:5000
seleniumgrid-web-1  * Running on https://127.0.0.1:5000
seleniumgrid-web-1 Press Ctrl+C to quit
seleniumgrid-web-1  * Restarting with stat
seleniumgrid-web-1  * Debugger is active!
seleniumgrid-web-1  * Debugger PIN: 545-765-375
seleniumgrid-web-1 177.19.0.1:5000 - - [09/Apr/2023 07:59:50] "GET / HTTP/1.1" 200 0.00
```

Figure 12.15: docker compose up

We can bring the container down using the command `docker compose down`, as shown in *Figure 12.16*:

```
[+] Running 2/1
  > Container seleniumgrid-web-1 Removed 0.2s
  > Network seleniumgrid_default Removed 0.2s
yogashivamathivanam/Yogashivam-MacBook-Pro SeleniumGrid %
```

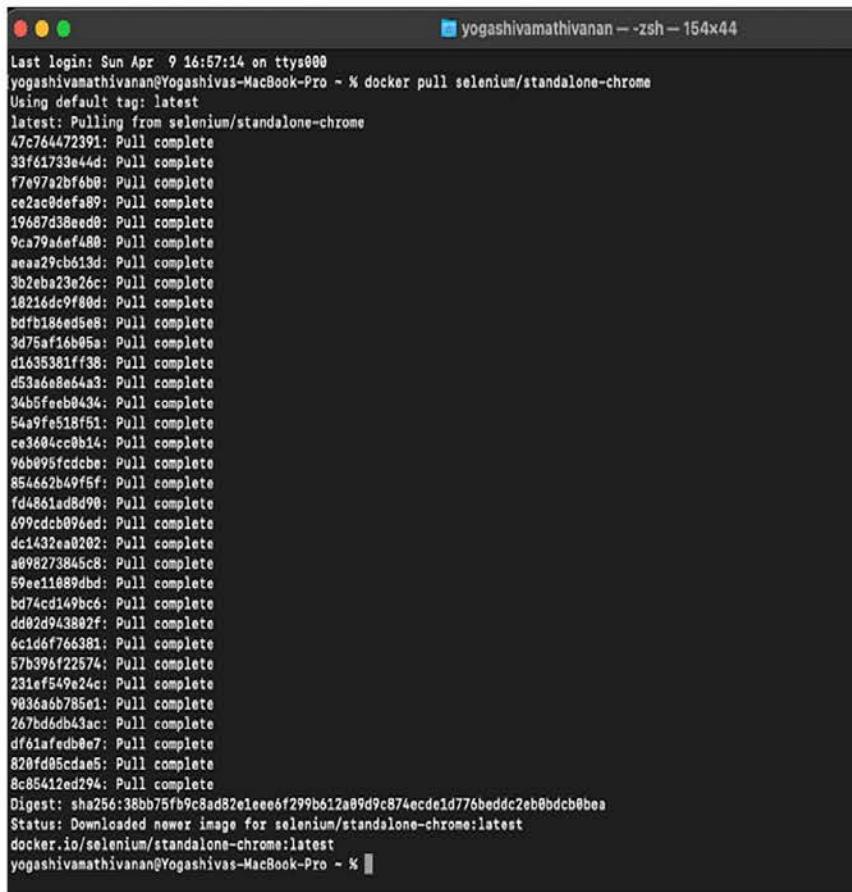
Figure 12.16: docker compose down

Running Selenium Tests with Docker

With the Docker knowledge, we are good to start running the Selenium test in Docker. Let us see the steps to run the Selenium test in the Chrome browser running inside a Docker container.

The first step is to pull the image from the docker hub registry before running a container based on the image. The command `docker pull selenium/standalone-chrome` downloads the latest version of the “selenium/standalone-chrome” image, as shown in *Figure 12.17*. This is a pre-built Docker image that includes the Chrome browser and the Selenium standalone server. It is a ready-to-use container for running Selenium tests on the Chrome browser. The “selenium/standalone-chrome” image

is maintained and updated by the Selenium project, and so you can be sure that you are getting a reliable and up-to-date version of Selenium with Chrome. To run the test in Firefox or Edge, we can use `docker pull selenium/standalone-firefox` and `docker pull selenium/standalone-edge` respectively.



```

yogashivamathivanan@Yogashivas-MacBook-Pro ~ % docker pull selenium/standalone-chrome
Using default tag: latest
latest: Pulling from selenium/standalone-chrome
47c764472391: Pull complete
33f61733e44d: Pull complete
f7e97a2bf6b9: Pull complete
ce2ac0defa89: Pull complete
19687d38eed9: Pull complete
9ca79a6ef480: Pull complete
aea29ccb613d: Pull complete
3b2eb323e26c: Pull complete
18216dc9f80d: Pull complete
bdfb18ed6d8e: Pull complete
3d75af16b05a: Pull complete
d1635381ff38: Pull complete
d53a6e8e64a3: Pull complete
34b5feeb8434: Pull complete
54a9fe518f51: Pull complete
ce3604cc0b14: Pull complete
96b095fcfdbe: Pull complete
854662b49f5f: Pull complete
fd4861ad8d90: Pull complete
699cdcb096ed: Pull complete
dc1432ea0202: Pull complete
a09827384c8: Pull complete
59ee110889dbd: Pull complete
bd74cd149bc6: Pull complete
dd02d943802f: Pull complete
6c1d6f766381: Pull complete
57b396f22574: Pull complete
231ef549e24c: Pull complete
9836a6b785e0: Pull complete
267bd6db43ac: Pull complete
df61afed8b07: Pull complete
820fd05cdae5: Pull complete
8c85412ed294: Pull complete
Digest: sha256:38bb75fb9c8ad82e1eee6f299b612a89d9c874ecde1d776beddc2eb0bdcb0bea
Status: Downloaded newer image for selenium/standalone-chrome:latest
docker.io/selenium/standalone-chrome:latest
yogashivamathivanan@Yogashivas-MacBook-Pro ~ %

```

Figure 12.17: `docker pull standalone-chrome`

For the next step, start the Docker container based on the “`selenium/standalone-chrome`” image using the command `docker run -d -p 4444:4444 -v /dev/shm:/dev/shm selenium/standalone-chrome`. The options are as follows:

- “`-d`” is used to run the container in detached mode, meaning it runs in the background and does not attach to the console.
- “`-p 4444:4444`” maps the container’s port 4444 to the host machine’s port 4444, allowing the Selenium server running inside the container to be accessed from the host machine.
- “`-v /dev/shm:/dev/shm`” mounts the shared memory directory on the host machine to the same directory inside the container, which is required for Chrome to function properly in a containerized environment.

- “`selenium/standalone-chrome`” specifies the name of the Docker image to use for the container.

This command will return the container ID in the response, as shown in *Figure 12.18*.

```
yogashivamathivanan@Yogashivas-MacBook-Pro ~ % docker run -d -p 4444:4444 -v /dev/shm:/dev/shm selenium/standalone-chrome
2fd19be0a408cb48b9fc9afa6181494a0899f651b786b64be5fb6b903308b
```

Figure 12.18: docker run standalone-chrome

We can see the docker container running in the docker application in *Figure 12.19*:

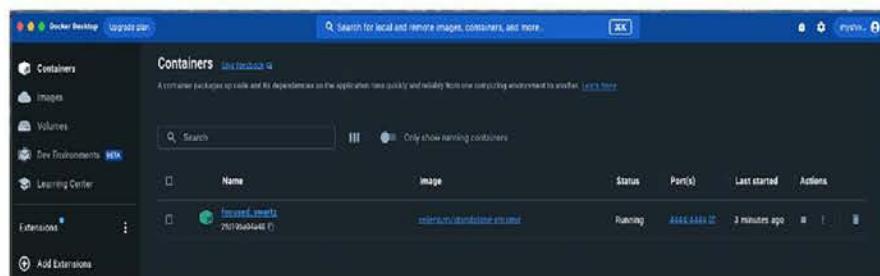


Figure 12.19: docker container

Furthermore, when we navigate to `http://localhost:4444` in the browser, we can see the Selenium Grid UI, as shown in *Figure 12.20*:

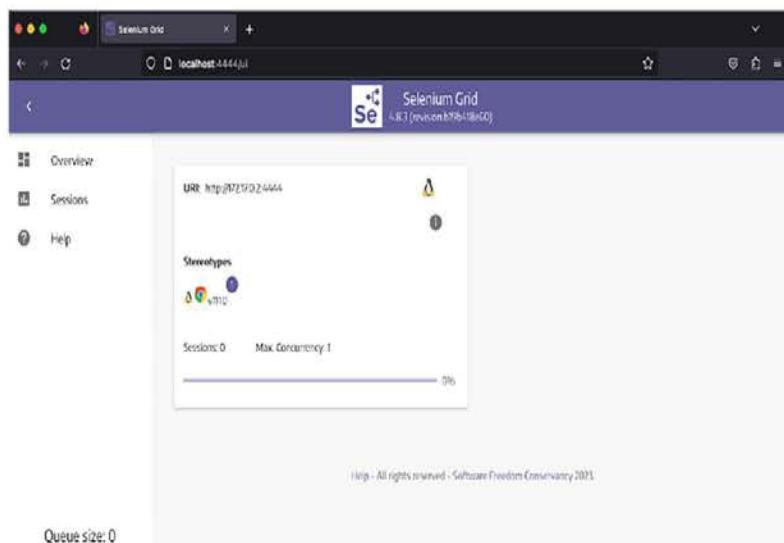


Figure 12.20: Selenium Grid UI from running the container

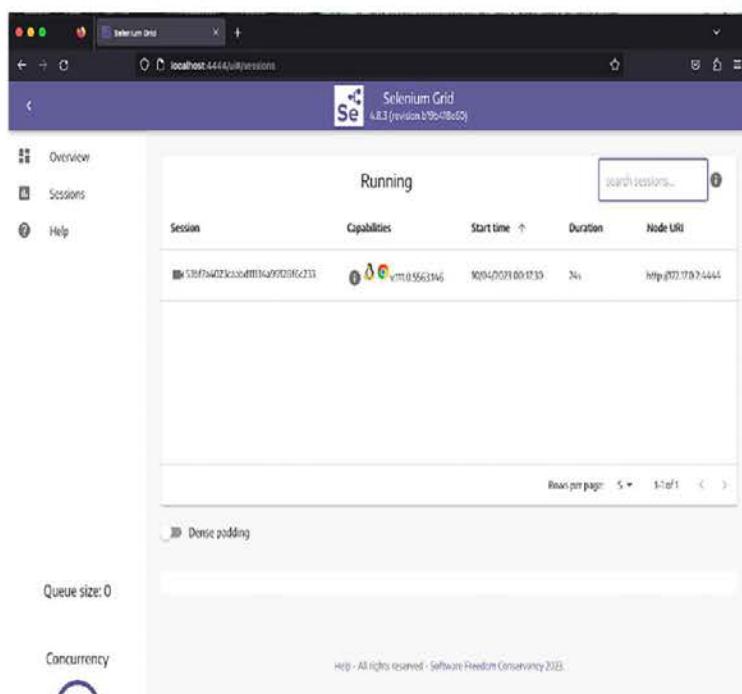
Lastly, let us create a Selenium test to run in the Chrome browser running inside a Docker container. In the following code, we configure the Chrome Options, connect to the Selenium server inside the docker container using `webdriver.Remote`, and specify the URL of the Selenium server `http://localhost:4444/wd/hub`. Here, we

have added 1000 second wait time to see the session on the server as in *Figure 12.21*.

CODE – SeleniumChromeTest

```
1. import time
2. from selenium import webdriver
3.
4. chrome_options = webdriver.ChromeOptions()
5. chrome_options.add_argument('--ignore-ssl-errors=yes')
6. chrome_options.add_argument('--ignore-certificate-errors')
7. chrome_driver = webdriver.Remote(
8.     command_executor='http://localhost:4444/wd/hub',
9.     options=chrome_options
10.)
11.
12. chrome_driver.get("https://www.google.com")
13. time.sleep(1000)
14. chrome_driver.quit()
```

Refer to the following figure:



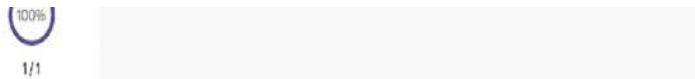


Figure 12.21: Selenium Chrome test session in Selenium Server

Selenium Grid

Selenium Grid is a tool that enables distributed testing of Selenium test scripts across multiple browsers and operating systems. It allows multiple machines to be used for running tests in parallel, which can significantly speed up the testing process and increase test coverage. This is done by routing commands to remote web browser instances, where one server acts as the hub. This hub routes test commands that are in JSON format, to multiple registered Grid nodes. Selenium Grid also allows for easy scaling of tests.

Hub enables simultaneous execution of tests on multiple machines, managing different browsers centrally, instead of conducting different tests for each of them. Selenium Grid makes cross-browser testing easy as a single test can be carried out on multiple machines and browsers, making it simple to analyze and compare the results.

Selenium Grid consists of two major components, a hub, and nodes, where the hub is the central point that manages the distribution of tests to the nodes, that execute the tests on different browser and OS configurations:

- **Selenium Grid Hub:** The hub receives test requests from the user and distributes them to the available nodes. It receives requests for test execution from the test scripts and routes them to the appropriate nodes, based on the capabilities of the nodes and the requirements of the tests. The hub also collects the results of the test execution and presents them to the user in a consolidated report.
- **Selenium Grid Nodes:** The Selenium grid nodes are the machines that execute the tests and send the results back to the hub. Each node has a specific set of capabilities that define the browser, operating system, and other settings that are required for the tests. Nodes can be added or removed from the Selenium Grid dynamically, depending on the needs. This allows us to scale the test execution across multiple machines and run tests in parallel.

Advantages of Selenium Grid

The Selenium Grid provides numerous advantages that can help improve automated testing processes. Following are some of the advantages of using Selenium Grid:

- **Parallel testing:** One of the primary advantages of using Selenium Grid is

the ability to run tests in parallel across multiple machines and browsers. This significantly reduces the time it takes to run a large number of tests, enabling faster feedback and a quicker release cycle.

- **Improved Test Coverage:** With Selenium Grid, you can easily run tests on multiple browsers and operating system combinations, which helps improve test coverage and ensures that your application is compatible with a wide range of environments.
- **Reduced Testing Time:** Selenium Grid allows you to distribute tests across multiple machines, which reduces testing time and helps you identify issues more quickly. Additionally, running tests in parallel allows you to test different parts of your application simultaneously, which can further reduce testing time.
- **Cost-effective:** Selenium Grid is an open-source tool, which means that there is no licensing fee involved in using it. Additionally, by using Selenium Grid, you can run tests on lower-cost machines and reduce the need for expensive hardware.
- **Better Resource Utilization:** With Selenium Grid, you can utilize idle machines in your network to run tests, which maximizes resource utilization and ensures that your testing infrastructure is being used efficiently.
- **Scalability:** Selenium Grid is highly scalable, which means that you can easily add more machines and browsers as your testing needs grow. This enables you to run more tests in parallel and ensures that your testing infrastructure can handle increased demand.
- **Flexibility:** Selenium Grid can be used with a variety of programming languages and test frameworks, which makes it a flexible tool that can be easily integrated into your existing testing processes.

Selenium Grid with Docker

Selenium Grid is a powerful tool that enables you to run automated tests across multiple machines and web browsers. It provides several advantages, including the ability to run tests in parallel, distribute tests across multiple machines, and reduce test execution time.

However, setting up and configuring Selenium Grid can be a complex and time-consuming process, particularly if you are not familiar with the underlying technologies, such as Java and JSON. It requires installing and configuring multiple

components, including the Selenium server, the hub, and the nodes, specifying browser configurations, setting up network connectivity, and ensuring compatibility between different versions of browsers and Selenium. Additionally, it can be difficult to manage the infrastructure required for Selenium Grid, including the maintenance and monitoring of multiple machines or browsers.

Despite the complexities involved, setting up and configuring Selenium Grid is worth the effort, especially if you have a large number of automated tests to run. The benefits of Selenium Grid, such as increased test efficiency, reduced testing time, and improved test coverage, outweigh the initial investment required to set up and configure the Grid.

This is where containerization technology comes in handy. By using containers to package the necessary software and dependencies for running tests, we can ensure consistency across all machines in the grid. Containers hosted on Docker provide a simplified and streamlined setup and configuration of Selenium Grid. Docker's ease of container creation, management, and scaling makes it simpler to deploy and manage Selenium Grid in a distributed environment. This, in turn, facilitates more efficient scaling of the grid to meet the needs of the testing process. Furthermore, Docker's quick spin-up and tear-down of containers enables parallel testing and faster completion of the testing process. Additionally, the lightweight and portable nature of Docker containers allows for seamless migration of Selenium Grid components between different environments."

Advantages of Selenium Grid with Docker

Selenium Grid allows the running of Selenium tests on multiple machines in parallel. Docker is a containerization platform that allows application packages and deploys as self-contained environments. The combination provides several benefits to automation engineers that simplify the setup and management of Selenium Grid in a distributed environment. Here are some of the advantages of using Selenium Grid with Docker:

- **Simplified setup and configuration:** By using Docker containers to host Selenium Grid components, the setup, and configuration of Selenium Grid can be simplified and streamlined. Docker allows for easy container creation, management, and scaling, thus making it easier to deploy and manage Selenium Grid in a distributed environment. This makes it easier to manage and maintain the infrastructure and allows for more efficient scaling of the grid to meet the needs of the testing process.
- **Consistency across all machines:** Docker helps to ensure that Selenium Grid is consistent across all machines in the grid. By packaging the necessary

software and dependencies for running tests into containers, developers can ensure that each machine in the grid has the same environment. This consistency ensures that tests run correctly and that results are reliable.

- **Faster testing:** Docker's ability to spin up and tear down containers quickly makes it possible to run tests in parallel and complete the testing process

faster. With Selenium Grid and Docker, developers can create as many containers as necessary and use them to run tests concurrently. This approach maximizes resource utilization and ensures that testing is completed as quickly as possible.

- **Scalability:** Docker is highly scalable, which means that Selenium Grid can be easily scaled up or down to meet the needs of the testing process. Developers can create and deploy additional containers as needed, to handle increased demand, and they can shut down containers when they are no longer needed. This ensures that resources are used efficiently and that Selenium Grid can handle any level of demand.
- **Portability:** Docker containers are lightweight and portable, allowing for easy migration of Selenium Grid components between different environments. This makes it easier to move Selenium Grid from one machine to another or from one cloud provider to another. Additionally, because Docker containers are self-contained environments, they can be easily backed up and restored as needed.
- **Cost savings:** Docker's ability to consolidate multiple applications onto a single server can help organizations save money on hardware and infrastructure costs. By using Docker to host Selenium Grid, developers can reduce the number of physical servers needed to run tests, which can lead to significant cost savings.

Running Selenium Grid in Docker

Let us look at the steps involved in setting up and running a Selenium Grid with Docker. Let us start with creating a `docker-compose.yaml` file that starts a Selenium Grid with three nodes: one for Chrome, one for Edge, and one for Firefox. The `docker-compose.yaml` file defines four services:

- “selenium-hub” starts the Selenium Hub.
- “chrome” starts a Chrome node and connects to the Selenium Hub.
- “edge” starts the Edge node and connects to the Selenium Hub.
- “firefox” starts the Firefox node and connects to the Selenium Hub.

Refer to the following code. “`version: "3.8"`” specifies the version of the Docker Compose file format being used, and “`services:`” is a list of services that are defined for the Docker Compose stack.

In the node services, the “`image`” field specifies the Docker image to be used, the “`shm_size`” field sets the shared memory size to 2GB, the “`depends_on`” field specifies

that this service depends on the selenium-hub service, and the “**environment**” field sets environment variables for the service, including the address of the event bus, which is the selenium-hub service.

In the selenium-hub service, the “**container_name**” field sets the name of the container to selenium-hub, and the “**ports**” field specifies the port mappings for the container, including ports 4442, 4443, and 4444.

CODE – docker-compose.yaml

```
1. version: "3.8"
2. services:
3.   chrome:
4.     image: selenium/node-chrome:latest
5.     shm_size: 2gb
6.     depends_on:
7.       - selenium-hub
8.     environment:
9.       - SE_EVENT_BUS_HOST=selenium-hub
10.      - SE_EVENT_BUS_PUBLISH_PORT=4442
11.      - SE_EVENT_BUS_SUBSCRIBE_PORT=4443
12.
13.   edge:
14.     image: selenium/node-edge:4.8.3-20230404
15.     shm_size: 2gb
16.     depends_on:
17.       - selenium-hub
18.     environment:
19.       - SE_EVENT_BUS_HOST=selenium-hub
20.       - SE_EVENT_BUS_PUBLISH_PORT=4442
21.       - SE_EVENT_BUS_SUBSCRIBE_PORT=4443
22.
23.   firefox:
24.     image: selenium/node-firefox:latest
25.     shm_size: 2gb
```



```

26.      depends_on:
27.        - selenium-hub
28.      environment:
29.        - SE_EVENT_BUS_HOST=selenium-hub
30.        - SE_EVENT_BUS_PUBLISH_PORT=4442
31.        - SE_EVENT_BUS_SUBSCRIBE_PORT=4443
32.
33.      selenium-hub:
34.        image: selenium/hub:latest
35.        container_name: selenium-hub
36.        ports:
37.          - "4442:4442"
38.          - "4443:4443"
39.          - "4444:4444"

```

This docker-compose file is used to create the Selenium Grid containers. We can use the command `docker compose up -d` to start the container. The images will be pulled if they do not already exist, and will be started, as shown in *Figure 12.22*:

```

SeleniumGrid -- zsh -- 96x28
yogashivamathivanan@Yogashivas-MacBook-Pro SeleniumGrid % docker compose up -d
[+] Running 44/4
✓ chrome 19 layers [██████████████████]    0B/0B    Pulled      54.6s
✓ edge 5 layers [███]    0B/0B    Pulled      66.8s
✓ firefox 3 layers [██]    0B/0B    Pulled      52.1s
✓ selenium-hub 13 layers [██████████████████]    0B/0B    Pulled      16.3s
[+] Running 1/2
✓ Network seleniumgrid_default  Created
[+] Running 1/2
[+] Running 1/2numgrid_default  Created
[+] Running 5/5grid_default  Created
Network seleniumgrid_default  Created
Container seleniumgrid-default  Started
Container selenium-hub  Started
✓ Container seleniumgrid-firefox-1  Started
✓ Container seleniumgrid-edge-1  Started
✓ Container seleniumgrid-chrome-1  Started
yogashivamathivanan@Yogashivas-MacBook-Pro SeleniumGrid %

```

Figure 12.22: Selenium Grid from docker compose

We can see the container in running status in the docker application in *Figure 12.23*:

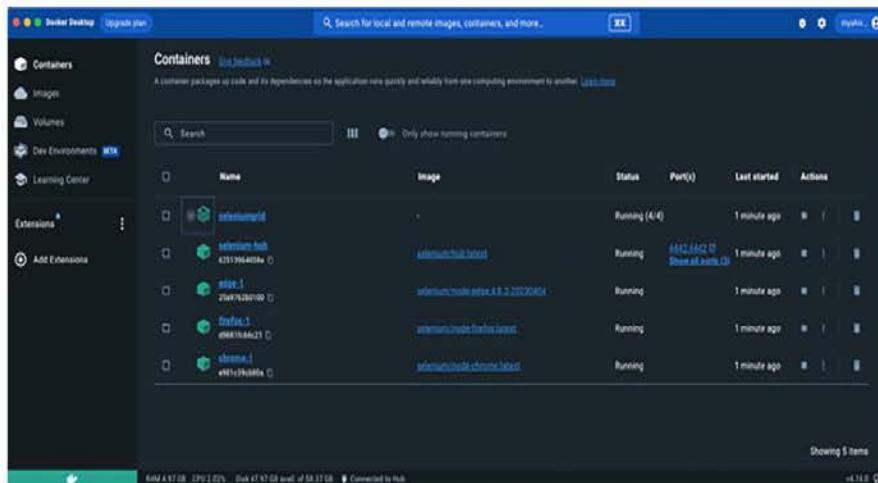


Figure 12.23: Selenium Grid running in Docker

Once the container is running, we can access the Selenium grid by going to `http://localhost:4444`, as shown in *Figure 12.24*:

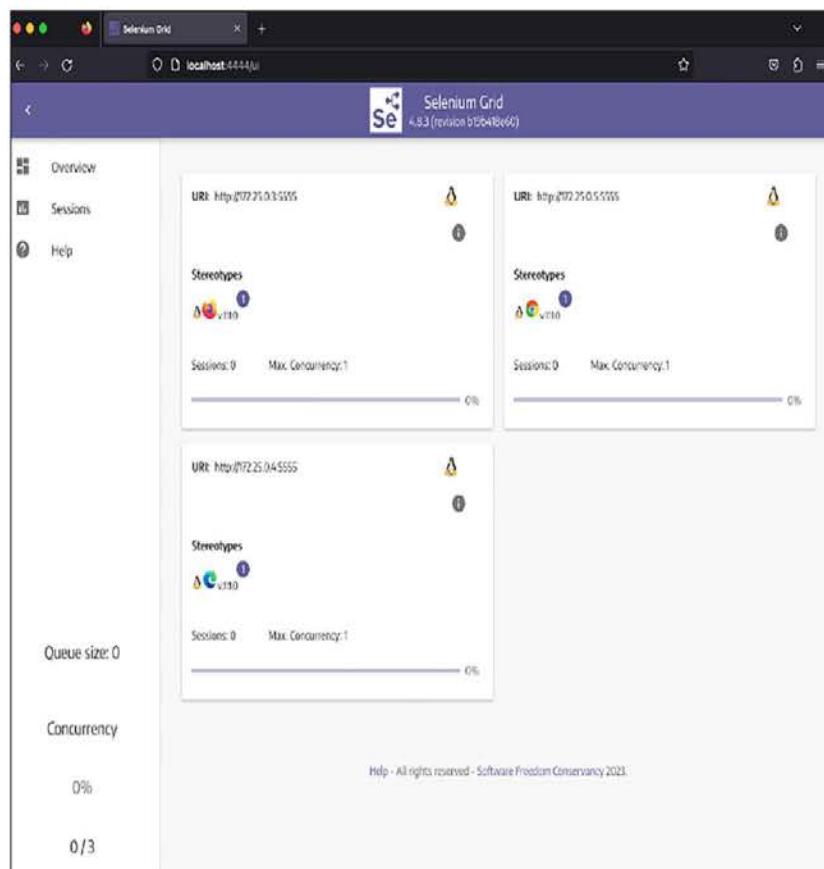


Figure 12.24: Selenium Grid Server

Executing parallel test

We can use **pytest-xdist** to execute the pytest in parallel, using the command “**pytest -v -s -n 6**”, as in *Figure 12.25*. The pytest command enables test runs in parallel and is configured to use up to 6 worker processes for parallel test execution. In the following simple test code, we have three tests each, to initiate and run in each browser. We have three of these files, and so there will be 9 tests run in parallel in three nodes. As we can see in *Figure 12.26* and *Figure 12.27*, all the nodes have 1 session and running in parallel. In *Figure 12.28*, we can see the 9 tests are successfully executed in 3 nodes.

CODE – test_Selenium.py

```
1. import time
2. from selenium import webdriver
3.
4.
5. class Test_Web():
6.     # Using Chrome
7.     def test_chrome(self):
8.         chrome_options = webdriver.ChromeOptions()
9.         chrome_options.add_argument('--ignore-ssl-errors=yes')
10.        chrome_options.add_argument('--ignore-certificate-errors')
11.        chrome_driver = webdriver.Remote(
12.            command_executor='http://localhost:4444/wd/hub',
13.            options=chrome_options
14.        )
15.
16.        chrome_driver.get("https://www.google.com")
17.        time.sleep(1000)
18.        chrome_driver.quit()
19.
```



```
20.  # Using Firefox
21. def test_firefox(self):
22.     firefox_options = webdriver.FirefoxOptions()
23.     firefox_options.add_argument('--ignore-ssl-errors=yes')
24.     firefox_options.add_argument('--ignore-certificate-errors')
25.     ff_driver = webdriver.Remote(
26.         command_executor='http://localhost:4444/wd/hub',
27.         options=firefox_options
28.     )
29.
30.     ff_driver.get("https://www.google.com")
31.     time.sleep(1000)
32.     ff_driver.quit()
33.
34. def test_edge(self):
35.     edge_options = webdriver.EdgeOptions()
36.     edge_options.add_argument('--ignore-ssl-errors=yes')
37.     edge_options.add_argument('--ignore-certificate-errors')
38.     edge_driver = webdriver.Remote(
39.         command_executor='http://localhost:4444/wd/hub',
40.         options=edge_options
41.     )
42.
43.     edge_driver.get("https://www.google.com")
44.     time.sleep(1000)
45.     edge_driver.quit()
```

Refer to Figure 12.25:

```
(ygnv) yogeshvamathivanan:Pycharm-Pro SeleniumGrid % pytest -v -s -n 6
=====
test session starts =====
v
platform darwin -- Python 3.10.7, pytest-7.3.4, pluggy-1.0.0 -- /Users/yogeshvamathivanan/PycharmProjects/SeleniumGrid/ygnv/bin/python
cachedir: .pytest_cache
rootdir: /Users/yogeshvamathivanan/PycharmProjects/SeleniumGrid
plugins: xdist-5.2.1
[py0] darwin Python 3.10.7 cmd: /Users/yogeshvamathivanan/PycharmProjects/SeleniumGrid
[py1] darwin Python 3.10.7 cmd: /Users/yogeshvamathivanan/PycharmProjects/SeleniumGrid
[py2] darwin Python 3.10.7 cmd: /Users/yogeshvamathivanan/PycharmProjects/SeleniumGrid
[py3] darwin Python 3.10.7 cmd: /Users/yogeshvamathivanan/PycharmProjects/SeleniumGrid
[py4] darwin Python 3.10.7 cmd: /Users/yogeshvamathivanan/PycharmProjects/SeleniumGrid
[py5] darwin Python 3.10.7 cmd: /Users/yogeshvamathivanan/PycharmProjects/SeleniumGrid
[py6] darwin Python 3.10.7 cmd: /Users/yogeshvamathivanan/PycharmProjects/SeleniumGrid
[py7] Python 3.10.7 (v3.10.7:acc0d13308, Sep  5 2022, 14:02:52) [Clang 13.0.0 (clang-1300.0.29.30)]
[py8] Python 3.10.7 (v3.10.7:acc0d13308, Sep  5 2022, 14:02:52) [Clang 13.0.0 (clang-1300.0.29.30)]
[py9] Python 3.10.7 (v3.10.7:acc0d13308, Sep  5 2022, 14:02:52) [Clang 13.0.0 (clang-1300.0.29.30)]
[py10] Python 3.10.7 (v3.10.7:acc0d13308, Sep  5 2022, 14:02:52) [Clang 13.0.0 (clang-1300.0.29.30)]
[py11] Python 3.10.7 (v3.10.7:acc0d13308, Sep  5 2022, 14:02:52) [Clang 13.0.0 (clang-1300.0.29.30)]
[py12] Python 3.10.7 (v3.10.7:acc0d13308, Sep  5 2022, 14:02:52) [Clang 13.0.0 (clang-1300.0.29.30)]
[py13] Python 3.10.7 (v3.10.7:acc0d13308, Sep  5 2022, 14:02:52) [Clang 13.0.0 (clang-1300.0.29.30)]
[py14] Python 3.10.7 (v3.10.7:acc0d13308, Sep  5 2022, 14:02:52) [Clang 13.0.0 (clang-1300.0.29.30)]
[py15] Python 3.10.7 (v3.10.7:acc0d13308, Sep  5 2022, 14:02:52) [Clang 13.0.0 (clang-1300.0.29.30)]
[py16] Python 3.10.7 (v3.10.7:acc0d13308, Sep  5 2022, 14:02:52) [Clang 13.0.0 (clang-1300.0.29.30)]
and [v] / pytest [v] / py3 [v] / py3 [v] / py3 [v] / py3 [v]
scheduling tests via LoadGridScheduling

test_Selenium.py::Test_Web::test_firefox
test_Selenium.py::Test_Web::test_chrome
test_Selenium.py::Test_Web::test_firefox
test_Selenium.py::Test_Web::test_chrome
test_Selenium2.py::Test_Web::test_chrome
test_Selenium2.py::Test_Web::test_chrome
```

Figure 12.25: Selenium Grid Parallel Test Execution

Refer to Figure 12.26:

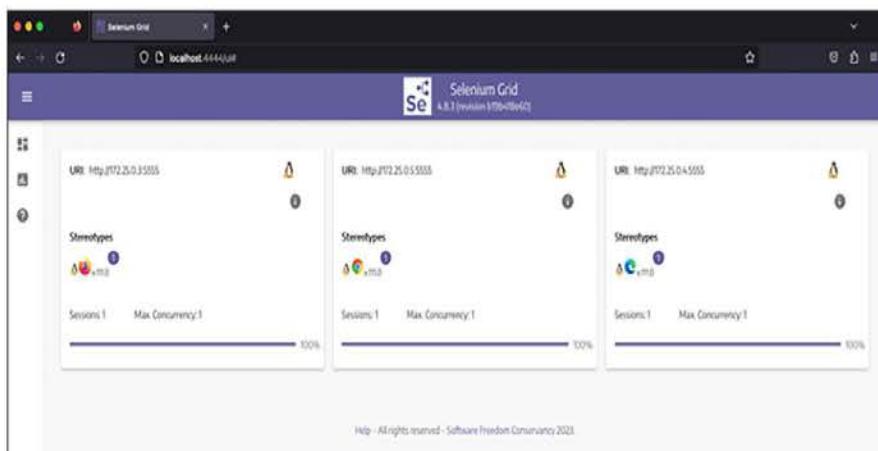


Figure 12.26: Selenium Grid Nodes and Session - I

Refer to Figure 12.27:

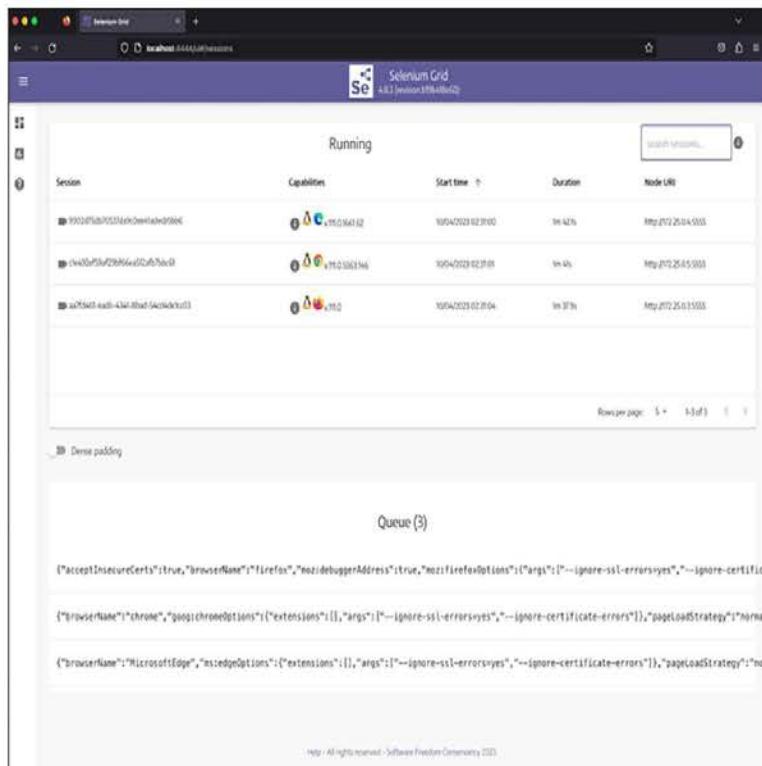


Figure 12.27: Selenium Grid Nodes and Session - II

Refer to Figure 12.28:

Figure 12.28: Successful Parallel Execution of Tests

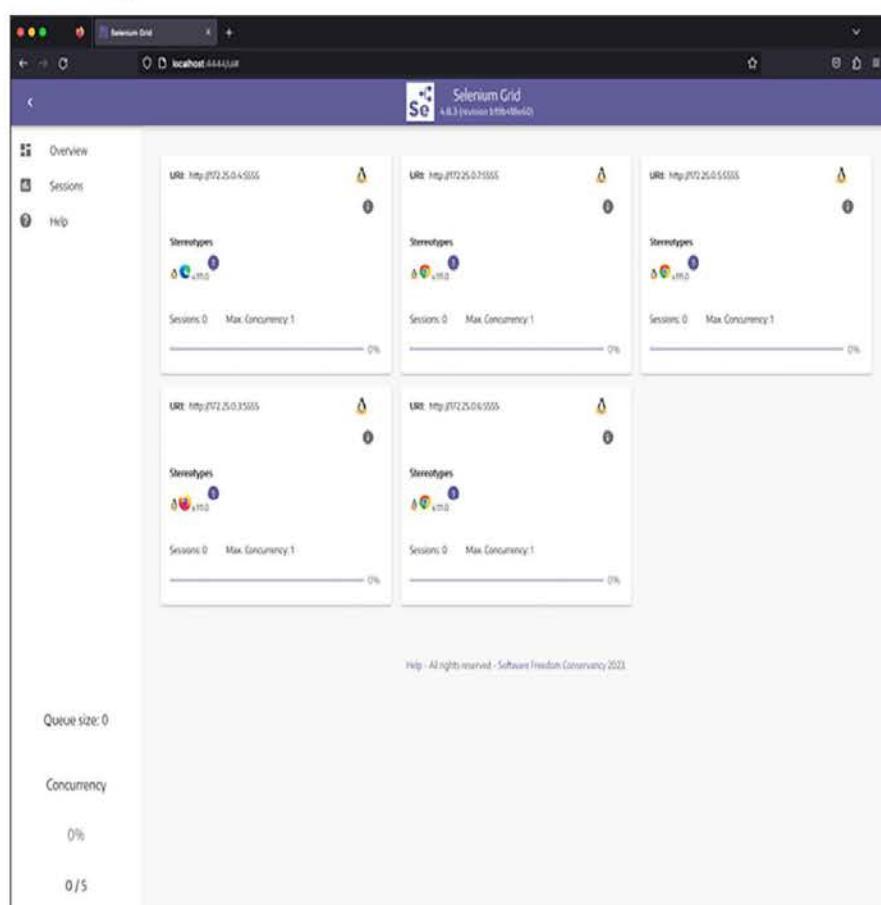
Scaling Number of Nodes

The number of services, in this case, the nodes, can be scaled up using the command `docker compose up --scale`. For instance, to scale the “chrome” service to 3 replicas, we can run the command “`docker compose up --scale chrome=3`” and this will start three instances of the “chrome” service. When we navigate to `http://localhost:4444`, we can see three Chrome nodes as in *Figure 12.30*. We can run the command “`docker ps -a`” to see that there are 3 instances of chrome running as in *Figure 12.29*:

```
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS               NAMES
cf4f952e1e93t      selenium/node-chrome:latest    "/opt/bin/entry_point.sh"   51 minutes ago   Up 51 minutes   5900/tcp            seleniumgrid-chrome-3
b736499a8922t      selenium/node-chrome:latest    "/opt/bin/entry_point.sh"   51 minutes ago   Up 51 minutes   5900/tcp            seleniumgrid-chrome-2
555fcd7e7f80t      selenium/node-chrome:latest    "/opt/bin/entry_point.sh"   51 minutes ago   Up 51 minutes   5900/tcp            seleniumgrid-chrome-1
d9881f66621t      selenium/node-firefox:latest    "/opt/bin/entry_point.sh"   2 hours ago     Up 59 minutes   5900/tcp            seleniumgrid-firefox-1
2ea976296100t      selenium/node-edge:v8.3-20230604t  "/opt/bin/entry_point.sh"   2 hours ago     Up 59 minutes   5900/tcp            seleniumgrid-edge-1
69539544684t      selenium/node-hub:latest       "/opt/bin/entry_point.sh"   2 hours ago     Up About an hour  8.8.8.8:4442-4442-4442-4442-4442/tcp  selenium-grid-hub
yogeshiventhivana/MacBook-Pro SeleniumGrid %
```

Figure 12.29: Docker List of Containers

Refer to *Figure 12.30*:



Conclusion

The Docker tool has become a vital tool in the modern software industry due to its ability to package and distribute applications in a consistent and efficient manner. It provides a lightweight and portable runtime environment, enabling developers to build, ship, and run applications quickly across multiple platforms. Docker also helps in overcoming the limitations of traditional virtualization by providing a more efficient way of sharing resources and reducing the overhead of virtualization. We looked at the various Docker terminologies such as images, containers, and Docker Hub in detail. Docker commands such as docker pull, docker build, docker run, docker ps, docker stop, docker rm, docker images, and docker push have also been explained, which help in managing Docker containers. Docker Compose is another important tool that helps in defining and running multi-container Docker applications. It simplifies the process of deploying and scaling applications by allowing developers to define the services, networks, and volumes in a single YAML file. We looked at running the Selenium tests in Docker, with its advantages such as faster test execution, easier test management, and better isolation. We also looked at how Selenium Grid enables distributed test execution across multiple nodes, and set up using Docker containers. We discussed the scaling the number of nodes. By leveraging the power of Docker and Selenium Grid, automation engineers can significantly reduce the time and effort required for testing web applications.

Key facts

- Virtualization allows multiple operating systems to run on a single physical machine. However, virtualization has some limitations, such as overhead, performance issues, and limited resource sharing.
- Docker is a containerization platform that allows packaging applications and dependencies into portable containers. Containers provide isolation, reproducibility, and scalability.
- Docker images are read-only templates used to create containers. Docker containers are lightweight, portable, and isolated environments that can run anywhere. Docker hub is a repository for sharing and storing Docker images.
- Basic docker commands are pull, build, run, ps, stop, rm, images, and push.
- Docker compose is a tool for defining and running multi-container Docker applications.
- Docker allows for easy and quick setup of test environments for running the

Selenium tests in docker, which can save time and resources.

- Selenium Grid allows for distributed testing across multiple machines and browsers. It has two components, the hub, and the nodes. The hub manages

the test queue and delegates tests to the nodes. The nodes execute the tests on a specific browser and operating system.

- Selenium Grid with docker provides better resource utilization and improved test efficiency. Docker compose can be used to define and run a Selenium grid cluster.
- Running Selenium Grid in Docker requires setting up the hub and nodes as separate containers. The hub and nodes must be configured to communicate with each other. Tests can be executed in parallel across multiple nodes for faster test execution.
- Docker Compose can be used to scale the number of nodes dynamically. Scaling the number of nodes can improve test execution speed and efficiency. Docker Compose up and down commands can be used to create and remove nodes dynamically.

Questions

1. What are the limitations of Virtualization compared to Docker? How does virtualization differ from containerization?
2. What is Docker? How are Docker images and Docker containers related to each other?
3. What is Docker Hub and how can it be used?
4. What is the purpose of the docker pull command? What is the difference between docker build and docker run commands? How do you stop and remove Docker containers?
5. What is Docker Compose? How can it be used to manage containers? What are its benefits?
6. How can Selenium tests be run in Docker containers?
7. What is Selenium Grid? How can be set it up in Docker containers?
8. How can the number of Selenium Grid nodes be scaled up or down?

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



CHAPTER 13

Bonus Chapter –

Python Interview

Questions

Introduction

We have discussed the significance of Python in automation and its growing demands. To help you prepare, this chapter is aimed at providing basic and intermediate-level Python programming interview questions specifically designed for Test Automation Engineers or **Software Development in Test (SDET)** roles. These questions cover interviews ranging from those for beginners and junior engineers, to those for senior engineers with up to 10 years of experience.

In this chapter, we will look only at the programming interview questions. For more theoretical and domain-related questions, follow and prepare the questions given at the end of each chapter for added confidence. These questions cover various aspects of Python programming skills that are tested during the interview. By going through these interview questions and their solutions, you will be able to strengthen your Python programming skills and feel more confident during your interview. So,

whether you are a beginner or an experienced Test Automation Engineer or SDET, make sure to brush up on these Python programming interview questions before your next interview to help you boost your confidence and increase your chances of landing the job. These questions are not presented in any order and are arranged randomly. It is important to note that the solutions provided are not the only possible

solutions, and there may be more optimized approaches that you can explore. Let us dive right into the programming questions.

Program 1

Write a program to return the count of each word in the provided Sentence.

CODE

```
1. def count_eachWord(sentence):
2.     # Split the sentence into words
3.     words = sentence.split()
4.
5.     # Create a dictionary to store the count of each word
6.     word_count = {}
7.
8.     # Loop through each word in the list
9.     for word in words:
10.         # If the word is already in the dictionary, increment its
11.         # count
12.         if word in word_count:
13.             word_count[word] += 1
14.         # If the word is not in the dictionary, add it with a
15.         # count of 1
16.         else:
17.             word_count[word] = 1
18.
19.     # Return the dictionary of word counts
20.
21. print(count_eachWord("This is a sample sentence to test the word
count program"))
```


OUTPUT

```
1. {'This': 1, 'is': 1, 'a': 1, 'sample': 1, 'sentence': 1, 'to': 1,
   'test': 1, 'the': 1, 'word': 1, 'count': 1, 'program': 1}
```

Program 2

Write a program to count each character in a sentence.

CODE

```
1. def char_count(input_string):
2.     char_count_dict = {}
3.     for char in input_string.lower():
4.         if char not in char_count_dict:
5.             char_count_dict[char] = 1
6.         else:
7.             char_count_dict[char] += 1
8.     print(char_count_dict)
9.
10.
11. char_count("Hello, world!")
```

OUTPUT

```
1. {'h': 1, 'e': 1, 'l': 3, 'o': 2, ',': 1, ' ': 1, 'w': 1, 'r': 1,
   'd': 1, '!': 1}
```

Program 3

Write a program to reverse a string, reverse each word in a given sentence, and reverse the complete sentence preserving the space and special characters.

CODE

```
1. def reverse_string_loop(string):
2.     reversed_str = ''
3.     for i in range(len(string) - 1, -1, -1):
4.         reversed_str += string[i]
5.     return reversed_str
```

```
6.  
7.  
8. def reverse_string_slicing(string):  
9.     return string[::-1]  
10.  
11.  
12. def reverse_string_reversed(string):  
13.     return ''.join(reversed(string))  
14.  
15.  
16. print(reverse_string_loop("hello world"))  
17. print(reverse_string_slicing("hello world"))  
18. print(reverse_string_reversed("hello world"))  
19.  
20.  
21. def reverse_words(sentence):  
22.     # Split the sentence into words  
23.     words = sentence.split()  
24.  
25.     # Reverse each word and join them back into a sentence  
26.     reversed_words = [word[::-1] for word in words]  
27.     reversed_sentence = ' '.join(reversed_words)  
28.  
29.     return reversed_sentence  
30.  
31.  
32. print(reverse_words("Hello world, && how are you?"))  
33.  
34.  
35. def reverse_sentence_preservingSpacesAndSpecialCharacters(sen-  
tence):  
36.     # convert the input string to a list of characters
```

```
37.     char_list = list(sentence)
38.
39.     # define two pointers i and j, pointing to the start and end
        of the list respectively
40.     i = 0
41.     j = len(char_list) - 1
42.
43.     # Loop until the pointers meet or cross each other
44.     while i < j:
45.         # if the character at index i is not a letter, move the
            pointer to the right
46.         if not char_list[i].isalpha():
47.             i += 1
48.         # if the character at index j is not a letter, move the
            pointer to the left
49.         elif not char_list[j].isalpha():
50.             j -= 1
51.         # if both characters at i and j are letters, swap them
            and move the pointers towards each other
52.         else:
53.             char_list[i], char_list[j] = char_list[j], char_list[i]
54.             i += 1
55.             j -= 1
56.
57.     # join the characters in the list back into a string
58.     return ''.join(char_list)
59.
60.
61. print(reverse_sentence_preservingSpacesAndSpecialCharacters("hel-
        lo, world! how are you today?"))
```

OUTPUT

```
1. dlrow olleh
   . . . . .
```

2. dlrow olleh

3. dlrow olleh
4. olleH ,dlrow && woh era ?uoy
5. yadot, uoyer! awo hdl row olleh?

Program 4

Write a program to find if a String and number is a Palindrome or not.

CODE

```
1. def is_palindrome_string(n):  
2.     if str(n) == str(n)[::-1]:  
3.         return True  
4.     else:  
5.         return False  
6.  
7.  
8. def is_palindrome_num(num):  
9.     original_num = num  
10.    reversed_num = 0  
11.  
12.    while num != 0:  
13.        reversed_num = num % 10 + (reversed_num * 10)  
14.        num //= 10  
15.  
16.    if original_num == reversed_num:  
17.        return True  
18.    else:  
19.        return False  
20.  
21.  
22.print(is_palindrome_string("tenet"))
```

```
23. print(is_palindrome_num(121))
```

OUTPUT

1. True
2. True

Program 5

Write a program to find the factorial of a number.

CODE

```
1. def factorialNum(num):  
2.     factorial = 1  
3.     if num < 0:  
4.         factorial = "Sorry, factorial does not exist for negative  
numbers"  
5.     elif num == 0:  
6.         print("The factorial of 0 is 1")  
7.     else:  
8.         for i in range(2, num + 1):  
9.             factorial = factorial * i  
10.    return factorial  
11.  
12.  
13. print("Factorial of", 12, "is", factorialNum(12))
```

OUTPUT

1. Factorial of 12 is 479001600

Program 6

Write a program to find if the number is prime or not and if the number is even or odd.

CODE

```
1. def is_prime(num):
```

```
2.     if num <= 1:  
3.         return False
```

```
4.     for i in range(2, int(num / 2) + 1):  
5.         if num % i == 0:  
6.             return False  
7.     return True  
8.  
9.  
10. def even_odd(num):  
11.     if num % 2 == 0:  
12.         print("even")  
13.     else:  
14.         print("odd")  
15.  
16.  
17. print(is_prime(59))  
18. even_odd(13)
```

OUTPUT

1. True
2. odd

Program 7

Write a program to find the second largest number in the list, to count the number of even and odd numbers in the list, sum all numbers in the list, to remove the duplicates, to reverse the list.

CODE

```
1. def find_second_largest(lst):  
2.     largest_num = lst[0]  
3.     second_largest_num = lst[0]  
4.     for num in lst:
```

```
5.         if num > largest_num:
6.             second_largest_num = largest_num
7.             largest_num = num
8.         elif num > second_largest_num and num != largest_num:
```

```
9.             second_largest_num = num
10.            return second_largest_num
11.
12.
13. def count_even_odd(lst):
14.     even_count = 0
15.     odd_count = 0
16.     for num in lst:
17.         if num % 2 == 0:
18.             even_count += 1
19.         else:
20.             odd_count += 1
21.     return even_count, odd_count
22.
23.
24. def find_sum(lst):
25.     total = 0
26.     for num in lst:
27.         total += num
28.     return total
29.
30.
31. def remove_duplicates(lst):
32.     # To remove duplicates from the list we can convert the list
33.     # to set.
34.     return list(set(lst))
35.
```

```
36. def reverse_list(lst):  
37.     left = 0  
38.     right = len(lst) - 1  
39.     while left < right:
```

374 ■ Selenium and Appium with Python

```
40.         lst[left], lst[right] = lst[right], lst[left]  
41.         left += 1  
42.         right -= 1  
43.     return lst  
44.  
45.  
46. list0fInt = [4, 5, 5, 9, 13, 45, 23, 9, 98, 13, 58]  
47. print(find_second_largest(list0fInt))  
48. print(count_even_odd(list0fInt))  
49. print(find_sum(list0fInt))  
50. print(remove_duplicates(list0fInt))  
51. print(reverse_list(list0fInt))
```

OUTPUT

1. 58
2. (3, 8)
3. 282
4. [98, 4, 5, 9, 13, 45, 23, 58]
5. [58, 13, 98, 9, 23, 45, 13, 9, 5, 5, 4]

Program 8

Write a program with an asterisk (*) before the parameter (*), which indicates that it is a variable-length argument. It allows the function to accept an arbitrary number of arguments.

CODE

```
1. def variable_length(size, *toppings, name=None):  
2.     print("Making Pizza of Size " + str(size))
```

```
3.     print(toppings)
4.     print(size + " " + name + " Pizza is ready with " + ,
      ".join[toppings])
5.
6.
```

```
7. variable_length("8 inch", "tomoto", "onion", "chicken", name="3
topping pizza")
```

OUTPUT

1. Making Pizza of Size 8 inch
2. ('tomoto', 'onion', 'chicken')
3. 8 inch 3 topping pizza Pizza is ready with tomoto, onion, chicken

Program 9

Write a program to return the longest sub-string without repeating, as well as the length of the string.

CODE

```
1. def longest_substring_without_repeating_chars(input_str):
2.     longest_substring = ""
3.     longest_substring_len = 0
4.     char_dict = {}
5.     for i in range(0, len(input_str)):
6.         if input_str[i] not in char_dict.keys():
7.             char_dict[input_str[i]] = i
8.         else:
9.             i = char_dict[input_str[i]]
10.            char_dict.clear()
11.            if longest_substring_len < len(char_dict):
12.                longest_substring_len = len(char_dict)
13.                longest_substring = str(char_dict.keys())
14.                print(longest_substring + "    " + str(longest_substring_len))
15.                return longest_substring_len
```

```
15. print(longest_substring_without_repeating_chars("ahghoihgglkv-  
16.  
17.  
18. print(longest_substring_without_repeating_chars("ahghoihgglkv-  
jqdiogfhqerwnoig"))
```

OUTPUT

```
1. dict_keys(['o', 'i', 'h', 'g', 'l', 'k', 'v', 'j', 'q', 'd'])    10  
2. 10
```

Program 10

Write a program to push all of a given number to the end of the list.

CODE

```
1. def push_num_to_end(lst, num):  
2.     count = 0  
3.     for i in range(len(lst)):  
4.         if lst[i] != num:  
5.             temp = lst[count]  
6.             lst[count] = lst[i]  
7.             lst[i] = temp  
8.             count += 1  
9.     return lst  
10.  
11.  
12. listWithInt = [4, 5, 5, 9, 13, 45, 23, 9, 98, 13, 58]  
13. print(push_num_to_end(listWithInt, 5))
```

OUTPUT

```
1. [4, 9, 13, 45, 23, 9, 98, 13, 58, 5, 5]
```

Program 11

Program 11

Write a program to find if two Strings are Anagrams or not.

CODE

```
1. def is_anagram(string1, string2):  
2.     # Convert strings to lists of lowercase characters  
3.     list1 = [char.lower() for char in string1]  
4.     list2 = [char.lower() for char in string2]  
5.
```

Bonus Chapter – Python Interview Questions ■ 377

```
6.     # Sort the lists and compare  
7.     if sorted(list1) == sorted(list2):  
8.         print("True")  
9.     else:  
10.        print("False")  
11.  
12.  
13.is_anagram("Listen", "Silent")
```

OUTPUT

```
1. True
```

Program 12

Write a program to find the intersection of two Strings using a set intersection, and using for loop and list comprehension.

CODE

```
1. def intersection_string(string1, string2):  
2.     intersection1 = set(string1.lower())  
3.     intersection2 = set(string2.lower())  
4.     common_chars = set()  
5.     print(intersection1.intersection(intersection2))  
6.     for i in range(len(string1.lower())):  
7.         for j in range(len(string2.lower())):  
8.             if string1[i].lower() == string2[j].lower():  
9.                 common_chars.add(string1[i])  
10.    print(common_chars)
```

```
9.         common_chars.add(string1[i].lower())
10.
11.     common_chars_list = [char for char in string2.lower() if char
12.                           in string1.lower()]
13.     print(str(common_chars))
14.
15.
16.intersection_string("Hello", "How are you")
```

OUTPUT

1. {'e', 'h', 'o'}
2. {'e', 'h', 'o'}
3. {'e', 'h', 'o'}

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



Index

Symbols	
<code>_name_ = “_main_”</code>	31
A	
absolute XPath	123
Acceptance Test Driven Development (ATDD)	224
action chains	139
for advanced operations	139, 140
advanced Web Page elements	
Alert popup	135
calendar date picker	136
checkboxes	132
dropdown	133
IFrame	134
radio button	132
Web Tables	138
window handles	137
Agile	13
Alert popup	135
Allure reporting tool	303
decorators	304, 305
installation	303, 304
screenshot	305
test execution	305-308
Alpha testing	9
Android Emulator	
configuring, on Mac and Windows	96-98
Android Hybrid App Automation	
error related to chrome version,	
troubleshooting	191
WebView context, switching to	189
WebView elements, identifying	186-188
Android keycodes	151, 152
APK File	
installing	98, 99
Appium	81, 82
Android application launching	103-105
Base Page and Screenshots	311
configurations	315
designing	309
Desired Capabilities	99-102
Driver initialization	310
folder structure	309, 310
implementation	310
installing, on Mac	92-95
installing, on Windows	86-92
Pages	317
Server Launch, programmatically	105, 106
Server Launch, with command prompt/terminal	106
test execution	320, 321
test execution and output	324
troubleshooting	325
utilities	315

Appium, for mobile automation
 advantages 85, 86
 architecture 83, 84
 drivers 85
 mobile application testing 82, 83
 Application Peripheral Interface 65
 Arithmetic operator 36
 assertions 174
 Python Assert Statement 175
 Assignment operator 37
 automating gestures 155
 long press gesture 157
 scroll gesture 158
 swipe gesture 157
 tap gesture 155
 automation framework 201
 behavior-driven development framework 206
 benefits 202, 203
 data-driven testing framework 205
 features 201, 202
 hybrid testing framework 207
 keyword-driven testing framework 206
 linear scripting framework 204
 modular testing framework 205
 selecting 207, 208
 types 204
 Automation testing 16, 17
 advantages 17
 Axes 126

B

Behave 243
 background 248
 data Parameterization 248
 data parameters 248
 environment and project setup 244, 245
 step definitions and implementation 245, 246
 test execution 247
 Behavior Driven Development (BDD) 224, 243
 keywords 243, 244

behavior-driven development framework 206
 advantages 206
 challenges 207

Beta testing 9
 Black-Box testing 9
 Boolean Data type 36
 Browser Drivers 65
 Bug 15

C

calendar date picker 136
 Cascading Style Sheets (CSS) 119
 Central Processing Unit (CPU) 28
 checkbox 132
 Chrome locators extensions 129
 Classes 54
 Comparison operator 37
 conditional statements 43
 IF-ELSE Condition 43
 conditional synchronization 165
 explicit wait 167
 Fluent wait 167
 implicit wait 165, 166
 Configparser 272
 configurations 271
 CSS selector
 ATTRIBUTE 121
 CLASS 120
 ID 119, 120
 MULTIPLE ATTRIBUTES 121
 relative CSS Selectors 122
 validating, in browser console 128
 CSV package 287

D

data-driven testing framework 205
 advantages 205
 challenges 206
 with data source 284, 285
 data types 34
 datetime 293

- datetime.date class 293
 datetime.datetime class 293
 datetime.time class 293
 datetime.timedelta class 293
 Defect/Bug Life Cycle 15
 states of defect 15, 16
 deprecated Selenium 3 code 75
 dictionary 42
 Docker 333
 advantages 334
 installation, on Mac 338, 339
 installation, on Windows 339-341
 Selenium Tests, running with 347-349
 terminology 335
 use cases, in software development 334, 335
 Docker commands 341
 docker build -t <image_name> 342
 docker images 344
 docker ps 343
 docker pull <image_name> 341
 docker push <image_name> 344
 docker rm <container_id> 344
 docker run <image_name> 343
 docker stop <container_id> 343
 Docker Compose 345-347
 Docker containers 335
 Dockerfile 336, 337
 Docker Hub 335
 Docker image 335
 Docker volume 344, 345
 Document Object Model (DOM) 113
 driver.close() function 77
 driver.quit() function 77
 dropdown 133
 dynamic wait 165
- E**
- element attributes/properties 152
 element locator strategy 144, 145
 element, finding by AccessibilityID 146, 147
- element, finding by class name 148
 element, finding by ID 148
 element, finding by name 149
 element, finding by UIAutomator 145, 146
 element, finding by Xpath 149
 elements methods
 finding 150
 end-to-end testing 9
 errors 51
 exception handling 52, 53
 exceptions 51
 explicit wait 167
- F**
- Faker 296
 fixtures, Pytest 215, 216
 parameterization 219
 reusing 216, 217
 Fluent wait 167
 FOR ELSE loop 46
 for loop 46
 functional testing 8
 functions 47
- G**
- global variables 47
 Graphical User Interface (GUI) 269
 grey-box testing 9
- H**
- Hybrid application 181, 182
 Android Hybrid App Automation 186
 IOS Hybrid App Automation 191
 mobile app examples 183
 multiple applications, switching 193, 194
 WebView 183-185
 hybrid testing framework 207
 advantages 207
 challenges 207
 Hypertext Transfer Protocol (HTTP) 64

I

- IF-ELSE Condition 43
- Iframes 134
- implicit wait 165
- inheritance 55
- integration testing 9
- Interactive Interpreter
 - using 28
- IOS Hybrid App Automation 191-193
- IOS Simulator Configuration
 - on Mac 107
- Iterative and Incremental model 12
 - advantages 13
 - application 12
 - disadvantages 13

J

- JavaScript Object Notation (JSON) 64
- JSON Wire Protocol 65

K

- keyword-driven testing framework 206
 - advantages 206
 - challenges 206

L

- linear scripting framework 204
 - advantages 204
 - challenges 204
- lists, Python 38, 39
- local variables 47
- locators 111
- locators, Selenium 112-115
 - element, locating using class name 116
 - element, locating using CSS selector 119
 - element, locating using ID 115
 - element, locating using link text 118
 - element, locating using name attribute 117
 - element, locating using partial link text 118
 - element, locating using tag name 118

- element, locating using XPath Locator 122
- logging 277, 278
 - formatter 279, 280
 - levels 278, 279
- Logical errors 51
- long press gesture 157
- loops 44
 - FOR ELSE loop 46
 - for loop 46
 - while loop 44

M

- Maneuvering Characteristics Augmentation System (CAS) 2
- manual testing 7
 - advantages 7
 - disadvantages 8
- markers, Pytest 213, 214
- miscellaneous driver methods 151
- modular testing framework 205
 - advantages 205
 - challenges 205
- modules 49, 50
- multiple web elements
 - finding 130

N

- non-functional testing 8
- NumPy 291

O

- Object-Oriented Programming (OOP) 55
- Objects 54
- Openpyxl 285
- operators, Python 36
 - Arithmetic operator 36
 - Assignment operator 37
 - Comparison operator 37
 - identity operator 38
 - logical operator 38

membership operator 38
 'os' module 273, 274

P

packages 50
 Page Object Model (POM) 253-255
 advantages, of folder structure 257
 best practices 267, 268
 creating 257-267
 features 255
 implementing 256
 Pandas 288, 290
 Pandas dataframe 289
 Pandas series 288, 289
 parallel test execution 283
 Pytest-xdist 283, 284
 'pathlib' module 277
 Pickle 291, 292, 293
 PyAutoGUI 269, 271
 PyCharm 27
 installation 27
 Pytest 208
 advantages 224
 executing, from PyCharm and
 Command Line 210-213
 fixtures 215, 216
 HTML reports, using pytest-html 222
 installing 208
 markers 213, 214
 soft assert 220, 222
 test, writing with 209
 pytest-html 222
 installing 223
 pytest-html report
 generating 223
 viewing 223
 Pytest-xdist 283
 Python 19
 advantages 21
 classes 54
 conditional statements 43

data types 34-36
 dictionaries 42
 functions 47
 indentation 43
 inheritance 55
 installing, in Mac OS 25, 26
 installing, in Windows 21-25
 lists 38
 loops 44
 modules 49
 objects 54
 operators 36
 requirement.txt 33
 sets 40
 Tuples 40
 variables 34-36
 virtual environment 31
 working with 21
 Python Assert Statement 175
 Python code
 executing 28
 executing, command line used 29
 executing, Interactive Interpreter used 28
 executing, with PyCharm 30
 Python interpreter 28
 Python logging module 278, 281, 282
 Python package 50
 Python Virtual Machine (PVM) 28

Q

quality assurance 5
 quality control 5

R

radio button 132, 133
 'random' package 295
 relative CSS Selectors 122
 relative XPath 123
 requirement.txt 33
 Robot framework 224
 advantages 239, 240

features 224, 225
 folder structure 229, 230
 folder structure best practices 230
 high-level architecture 225
 installation 226
 keywords 228, 229
 libraries 227, 228
 logs and reports 238, 239
 test data management 231-233
 test execution flow 233
 Robotic Process Automation (RPA) 224

S

sanity testing 9
 screenshots 268, 269
 scroll gesture 158
 SCRUM Agile methodology 13, 14
 advantages 14
 application 14
 disadvantages 14
 Selenium 59-61
 advantages 62, 63
 browser invoke 72, 73
 browser window setup 74
 exception handling 172, 173
 exceptions 170-172
 locators 112
 login scenario, automating 75
 versus other testing tools 63
 Selenium 3 architecture 64, 65
 Selenium 4 architecture 66
 benefits 67
 Selenium 4 test script 76
 Selenium API 65
 Selenium Client Library 65
 Selenium commands
 finding 130
 Selenium Grid 62, 329, 351
 advantages 351-354
 number of nodes, scaling 362
 parallel test, executing 358-361

running, in Docker 354-357
 with Docker 352, 353
 Selenium Grid Hub 351
 Selenium Grid Nodes 351
 Selenium Integrated Environment (IDE) 61
 Selenium Remote Control (RC) 60, 61
 Selenium Tests
 running, with Docker 347-350
 Selenium Webdriver 61
 architecture 63
 web applications layers 64
 Selenium WebDriver
 installing 67-71
 set 40
 Simulator Application
 launching 108, 109
 smoke testing 10
 Software Development in Test (SDET) 365
 Software Development Life Cycle (SDLC) 10
 Iterative and Incremental model 12
 SCRUM Agile methodology 13
 Waterfall model 10
 software testing 1
 automation 16, 17
 benefits 3
 concepts 4
 functional testing 8
 manual testing 7
 non-functional testing 8
 quality assurance 5
 quality control 5
 significance 2, 3
 types 6
 software testing process
 test case 6
 test closure 6
 test execution 6
 test plan 6
 test scenario 6
 test strategy 6
 static wait 165

swipe gesture 157
 synchronization 165
 conditional synchronization 165
 unconditional synchronization 165
 Syntax errors 51
 System testing 9

T

tap gesture 155
 test execution flow, Robot framework
 test case execution 234, 235
 test setup/test suite setup 233, 234
 test teardown/test suite teardown 235
 Webdrivermanager 234
 testing frameworks
 comparison 249
 Tuples 40

U

unconditional synchronization 165
 unittest framework 240, 241, 243
 unit testing 8
 user acceptance testing 9

V

variable data type 35
 variables 34
 virtual environment, Python
 creating 31, 32
 virtualization 331
 limitations 331, 332

W

Waterfall model 10, 11
 advantages 11
 application 11
 disadvantages 11, 12
 web application layers, Selenium
 Business Logic Layer (BLL) 64
 Data Access Layer (DAL) 64
 Data Service Layer (DSL) 64
 Presentation Layer (PL) 64
 Webdriver Manager 77
 versus, driver path 78, 79
 web elements, in DOM 113, 114
 web tables 138
 WebView 183-185
 while loop 44
 White-Box testing 9
 World Wide Web Consortium (W3C)
 Selenium protocol 65

X

XPath 123
 absolute XPath 123
 Axes 126, 127
 relative XPath 123
 validating, in browser console 128
 with logical operators 124
 with XPath Functions 124

Selenium and Appium with Python

DESCRIPTION

Appium and Selenium are popular open-source frameworks widely used for test automation in the software industry. Python, on the other hand, is a versatile and powerful programming language known for its simplicity and readability. Combining Appium and Selenium with Python offers numerous advantages for test automation, including a simplified testing process, faster test execution, and increased efficiency in test script development.

Written by a Test Automation Architect, this book aims to enhance your knowledge of Selenium and Appium automation tools. The book will help you learn how to leverage Python for test automation development, gaining skills to automate various types of elements, actions, gestures, and more in web and mobile applications, including Android and iOS. Furthermore, the book will help you create a robust and maintainable test automation framework from scratch. Lastly, the book will teach you how to utilize Selenium Grid with Docker to run and distribute tests across multiple machines, enabling you to maximize efficiency and productivity in test automation.

By the end of the book, you will be able to build effective and scalable automated testing solutions using Python.

KEY FEATURES

- Get started with automation testing using Python, Selenium, and Appium.
- Learn how to create a test automation framework from scratch
- Learn how to perform web and mobile app testing using Selenium and Appium, respectively.

WHAT YOU WILL LEARN

- Learn how to automate web testing with Selenium and Python.
- Learn how to automate Mobile testing with appium and Python.
- Learn how to handle exceptions and synchronization for web and mobile apps.
- Learn how to automate Hybrid apps using Selenium and Appium.
- Learn how to integrate Selenium Grid with Docker.

WHO THIS BOOK IS FOR

This book is for Software Quality Assurance, including Test Automation Engineers, Product Owners, and Developers who are looking to enhance their test automation skills.



BPB PUBLICATIONS

www.bpbonline.com

ISBN 978-93-5551-835-4



9 789355 518354

